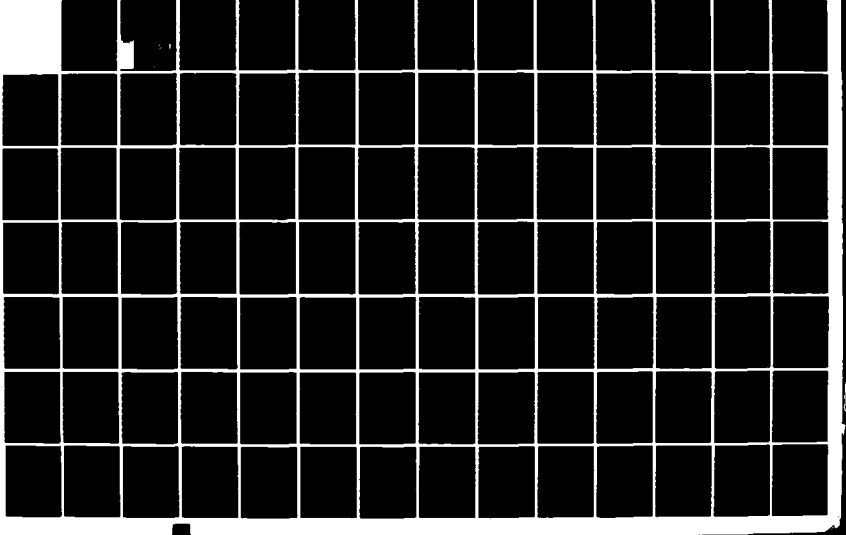
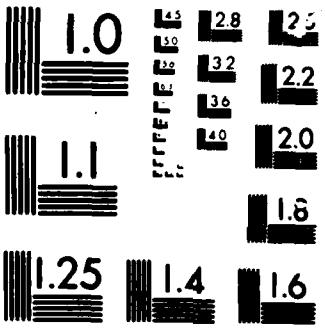
AD-A167 315 DISTRIBUTED COMPUTING FOR SIGNAL PROCESSING: MODELING OF ASYNCHRONOUS PAR..(U) PURDUE UNIV LAFAYETTE IN SCHOOL OF ELECTRICAL ENGINEERING L H JAMIESON ET AL. 1/2

UNCLASSIFIED MAR 86 ARO-18790.17-EL DAAG29-82-K-0101 F/G 9/2 NL





MICROCOPY CHART

Distributed Computing for Signal Processing: Modeling of Asynchronous Parallel Computation

Final Report

L.H. Jamieson, H.J. Siegel, P.H. Swain,
G.B. Adams III, J.T.Kuehn,
W.E. Kuhn III, G-M. Lin, R.J. McMillen,
T.A. Rice, R.R. Seban, B.W. Smith,
K.D. Smith, D.L. Tuomenoksa

March 1986

U.S. Army Research Office
Contract No. DAAG29-82-K-0101

School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. N/A	3. RECIPIENT'S CATALOG NUMBER N/A
4. TITLE (and Subtitle) Distributed Computing for Signal Processing: Modeling of Asynchronous Parallel Computation Final Report		5. TYPE OF REPORT & PERIOD COVERED Final Report: April 1, 1982 September 30, 1985
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) L.H. Jamieson, H.J. Siegel, P.H. Swain, G.B. Adams III, J.T. Kuehn, W.E.Kuhn III, G.-M. Lin, R.J. McMillen, T.A. Rice, R.R. Seban, B.W. Smith, K. D. Smith, D.L. Tuomenoksa		8. CONTRACT OR GRANT NUMBER(s) Contract No. DAAG29-82-K-0101
9. PERFORMING ORGANIZATION NAME AND ADDRESS School of Electrical Engineering Purdue University West Lafayette, IN 47907		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS U. S. Army Research Office Post Office Box 12211 Research Triangle Park, NC 27709		12. REPORT DATE March 1986
		13. NUMBER OF PAGES 168
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

NA

18. SUPPLEMENTARY NOTES

The view, opinions, and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other documentation.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

distributed computing, parallel processing, asynchronous computation, signal processing

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report compiles the results of the research performed under U.S. Army Research Office Contract Number DAAG29-82-K-0101, covering the period April 1, 1982 through September 30, 1985. The work is in the area of modeling asynchronous parallel architectures and computation for applications in the areas of digital image and signal processing. The work can be broadly divided into three areas:

1. Case studies of parallel image processing algorithms and tasks, the

(continued on next page)

objective of which is to study the interaction of parallel processes and parallel architectures. These are described in Papers 1 through 5 and in portions of Appendices A, B, C, and D.

2. Modeling of interconnection networks. An important component of any large-scale distributed system is the interconnection network. Different techniques for modeling interconnection networks were developed and are described in Papers 6 through 9 and in portions of Appendices A, C, and E.

3. Aspects of the problem of modeling parallel processes and parallel architectures. This includes mechanisms for describing MIMD algorithms (Paper 11 and portions of Appendices A and D), application of a Petri net based modeling scheme to SIMD and pipeline implementations of example image processing algorithms (portions of Appendices A and F), consideration of performance criteria for parallel image processing algorithms (portions of Appendix F), matching algorithms with macropipelined distributed systems (Paper 12 and portions of Appendix G), new models for the organization of distributed systems comprised of collections of special purpose computing devices (Paper 10), and companion features for describing parallel processes and parallel architectures (portions of Appendices A and D).

DISTRIBUTED COMPUTING FOR SIGNAL PROCESSING: MODELING OF ASYNCHRONOUS PARALLEL COMPUTATION

FINAL REPORT

L.H. Jamieson, H.J. Siegel, P.H. Swain,
G.B. Adams III, J.T. Kuehn, W.E. Kuhn III,
G.-M. Lin, R.J. McMillen, T.A. Rice, R.R. Seban,
B.W. Smith, K.D. Smith, D.L. Tuomenoksa

March 1986

U.S. Army Research Office
Contract No. DAAG29-82-K-0101

School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907

Table of Contents

Paper #7

Robert R. Seban and Howard Jay Siegel, "Analysis of Partitionability Properties of Topologically Arbitrary Interconnection Networks," *Proceedings of the Fifth International Conference on Distributed Computer Systems* (IEEE Catalog No. 85CH2149-3), pp. 173-181, Denver, Colorado, May 1985

Paper #8

Robert J. McMillen and Howard Jay Siegel, "Evaluation of Cube and Data Manipulator Networks," *Journal of Parallel and Distributed Computing*, Vol. 2, No. 1, pp. 79-107, February 1985

Paper #9

George B. Adams III and Howard Jay Siegel, "A Survey of Fault-Toleran Multistage Networks and Comparison to the Extra Stage Cube," *Proceedings of the Seventeenth Hawaii International Conference on System Sciences*, pp. 268-277, Honolulu, Hawaii, January 1984

Paper #10

Veljko M. Milutinovic, J. J. Crnkovic, L. Y. Chang, and Howard Jay Siegel, "The LOCO Approach to Task Allocation AIDA by VERDI," *Proceedings of the Fifth International Conference on Distributed Computer Systems* (IEEE Catalog No. 85CH2149-3), pp. 359-368, Denver, Colorado, May 1985

Paper #11

Kirk D. Smith and Leah H. Jamieson, "MIMD Algorithm Analysis: Low Level Algorithm Descriptions," *Fifth International Conference on Distributed Computing Systems* (IEEE Catalog No. 85CH2149-3), pp. 150-157, Denver, Colorado, May 1985

Paper #12

Bradley W. Smith and Howard Jay Siegel, "Models for Use in the Design of Macro-Pipelined Parallel Processors," *Proceedings of the Twelfth Annual International Symposium on Computer Architecture* (IEEE Catalog No. 85CH2144-4), pp. 116-123, Boston, Massachusetts, June 1985

Appendix A

Distributed Computing for Signal Processing: Modeling of Asynchronous Parallel Computation 1983 Progress Report, Purdue Electrical Engineering Technical Report TR-EE 83-11, March 1983

Appendix B

Design of the Operating System for the PASM Parallel Processing System, Ph.D. Thesis by David Lee Tuomenoksa, Faculty Advisor: Howard Jay Siegel

Appendix C

Fault Tolerant Interconnection Networks and Image Processing Applications for the PASM Parallel Processing Systems, Ph.D. Thesis by George B. Adams III, Faculty Advisor: Howard Jay Siegel

Appendix D

Analysis of MIMD Algorithms: Features, Measurements, and Results, Ph.D. Thesis by Kirk D. Smith, Faculty Advisor: Leah H. Jamieson

Appendix E

Topological Properties of Interconnection Networks for Parallel Processors, Ph.D. Thesis by Robert R. Seban, Faculty Advisor: Howard Jay Siegel

Appendix F

Studies in Parallel Image Processing, Ph.D. Thesis by Gie-Ming Lin, Faculty Advisor: Philip H. Swain; Purdue Electrical Engineering Technical Report TR-EE 84-29, August 1984

Appendix G

On the Design and Modeling of Special Purpose Parallel Processing Systems, Ph.D. Thesis by Bradley Warren Smith, Faculty Advisor: Howard Jay Siegel

INTRODUCTION

This report compiles the results of the research performed under U.S. Army Research Office Contract Number DAAG29-82-K-0101, covering the period April 1, 1982 through September 31, 1985. The work is in the area of modeling asynchronous parallel architectures and computation for applications in the areas of digital image and signal processing. The work can be broadly divided into three areas:

1. Case studies of parallel image processing algorithms and tasks, the objective of which is to study the interaction of parallel processes and parallel architectures. These are described in Papers 1 through 5 and in portions of Appendices A, B, C, and D.
2. Modeling of interconnection networks. An important component of any large-scale distributed system is the interconnection network. Different techniques for modeling interconnection networks were developed and are described in Papers 6 through 9 and in portions of Appendices A, C, and E.
3. Aspects of the problem of modeling parallel processes and parallel architectures. This includes mechanisms for describing MIMD algorithms (Paper 11 and portions of Appendices A and D), application of a Petri net based modeling scheme to SIMD and pipeline implementations of example image processing algorithms (portions of Appendices A and F), consideration of performance criteria for parallel image processing algorithms (portions of Appendix F), matching algorithms with macropipelined distributed systems (Paper 12 and portions of Appendix G), new models for the organization of distributed systems comprised of collections of special purpose computing devices (Paper 10), and companion features for describing parallel processes and parallel architectures (portions of Appendices A and D).

Paper 1

Parallel Algorithms for Computer Vision

From the Proceedings of the 1983 Computer Society Workshop on Computer Architecture for Pattern Analysis and Image Database Management, Pasadena, CA, October, 1983.

PARALLEL ALGORITHMS FOR COMPUTER VISION

Thomas A. Rice
Leah Jamieson Siegel

School of Electrical Engineering
Purdue University
West Lafayette, IN 47907

Abstract

An application of parallel processing to the computationally intensive task of computer vision is presented. Computational speedups, both theroretical and experimental, are derived and presented for the extraction of several parameters based upon the SRI vision module and Fourier descriptors. Good results are obtained for moderate numbers of processing elements. The use of parallel processing allows easier expansion and modification of the vision algorithms as compared with a hardware approach.

1. Introduction

Parallel processing offers the potential of providing fast, flexible solutions to many computationally intensive tasks. In this paper, the use of parallelism for computer vision is described. Theoretical analyses and simulation results are presented. Considerations for the design of a parallel architecture for computer vision are discussed.

2. Definitions for the Parallel Simulation

In this section, two general models of parallel computation are defined and the specific model used for the computer vision task is presented. The implementation of the parallel simulation is described.

Model

Single instruction stream - multiple data stream (SIMD) machines [4] represent a form of synchronous, highly parallel processing. Systems with up to 1,000 full processors have been proposed [10, 14]; systems with as many as 9,000 and 16,000 simple processors have been built [2, 3]. An SIMD machine typically consists of a *control unit*, a set of P *processing elements (PEs)*, each a processor with its own memory, and an *interconnection network*. The control unit broadcasts instructions to all PEs and each active PE executes the instruction on the data in its own memory. The interconnection network allows data to be transferred among the PEs. SIMD machines are especially well-suited for exploiting the parallelism inherent in certain tasks performed on vectors and arrays.

Multiple instruction stream - multiple data stream (MIMD) machines [4] represent asynchronous parallel processing. MIMD systems with 16 [18] and 50 [16] processors have been built; MIMD systems with as many as 4,000 processors [6] have been proposed. An MIMD machine typically consists of P processors and M memories, $M \geq P$, where each processor can follow an independent instruction stream. As with SIMD machines, there is a multiple data stream and an interconnection network. Thus, there are P independent processors that can communicate among themselves. There may be a coordintor unit to oversee the activities of the processors.

This research was supported by the United States Army Research Office, Department of the Army, under grant number DAAG29-82-K-0101.

The parallel machine model assumed for the computer vision task consists of a set of PEs under the management of a control unit. The number of PEs is a power of two: $N=2^n$. Each of the PEs has a unique "address" between 0 and N - 1. In addition, there exists an interconnection network to allow the simultaneous transfer of data among the PEs. For the computer vision task, the transfer patterns required will be uniform modulo shifts and "cube" interconnection functions. In a *uniform modulo shift*, PE j transfers data to PE (j+d) modulo N for all j, $0 \leq j < N$, given a positive or negative integer distance d. The value of d may vary from one transfer to the next; however, for a given transfer, all PEs will send their data the same distance d. The set of *cube* interconnection functions consists of $n = \log_2 N$ functions, $cube_i$, for $0 \leq i < n$ [13]. If $p_{n-1} \cdots p_i \cdots p_0$ is the binary representation of a PE's address, then the $cube_i$ function exchanges data between all pairs of PEs whose addresses differ in bit i:

$$cube_i(p_{n-1} \cdots p_i \cdots p_0) = p_{n-1} \cdots \bar{p}_i \cdots p_0.$$

The model assumed here combines SIMD and MIMD attributes. Each PE will contain the same code but will execute the code on a different subimage. However, within each PE, the code can run in MIMD mode. This modification to the basic models allows faster execution on some code than a pure SIMD model without incurring the expense of the full flexibility of an MIMD machine. The gains in speed will occur on the execution of conditional statements:

where <condition> do <block1>
elsewhere do <block2>

In SIMD mode, those PEs satisfying the <condition> execute <block1>. Then the remaining PEs execute <block2>. In the model here, <block1> and <block2> will be executed concurrently, but in different sets of PEs. On the other hand, this is not full MIMD mode, as it is required that the code in each PE be the same. This aids in insuring synchronization and thus helps enforce data coherence, e.g., insuring that a PE acquires the correct version of a variable from another PE.

Synchronization can take place in one of two ways. First, synchronization is required at all data transfer points. This is done because data transfers often involve

the same variable for all of the PEs. Thus, it does not matter if the separate processors take different times to execute their code, as they will be forced to synchronize at transfers to insure coherence. Explicit synchronization will also be possible by one of the simulation language constructs that requires that all PEs finish a section of code before any can move to the next section of code.

The motivation for the assumed model comes about from two directions. First, for many image processing operations, it is natural to consider executing the same code on subimages of the original image. Each subimage is a valid image and the same types of operations are needed on the pixels of each subimage. Second, since the actual quantities of the various operations that will be performed on each subimage may vary, asynchronous operation may allow higher PE utilization.

Simulation

There are two major approaches to the development of parallel software. Either the software can be of a generally descriptive nature to illustrate the parallelism (or lack thereof) inherent in a task, or the software can be designed to be compilable and testable, either by parallel execution or via serial simulation. Due to the computational intensity and intricacy of the computer vision task, the most reliable way to insure correctness is via testing. This will guarantee that typical problem cases are being handled correctly by testing the software for a variety of images. A set of test images, some with multiple objects, was used for debugging and for analyzing computational speedup. Therefore, the software was designed so that it could be compiled and tested.

Programming was done in a modified version of the 'C' language [7]. This language was chosen for the capabilities it provides for developing parallel data structures and the high degree to which one can manipulate system information (such as memory areas). The latter played a large part in the simulating of parallel data transfers. The actual conversion of the serial 'C' language to a parallel language was done via macros and support subroutines. These features were designed to facilitate the development of parallel code without requiring the user to know the specific details of the serial implementation. Thus one can simply use the macro file without knowing its details and can then write parallel code.

The major points of this implementation are as follows. A construct of the form

in_pe { codeblock; }

executes the enclosed block of code in each of the PEs. Variables "in" prepended to a variable indicates that the variable is local to a PE. All other variables are assumed to be global (i.e., the control unit has one copy of the variable). Global variables are used for such operations as loop control and overall conditional testing. There are also versions of the "in_pe" construct that allow the code to be executed in a limited subset of the PEs. These schemes use an address mask [12], which is a matching format that the PE address must match for execution to occur in that PE.

Interprocessor communication is accomplished via a "transfer" subroutine:

transfer(destination_address,source_address,offset).

The transfer routine uses these addresses along with information about the size and structure of the PE data space to simulate the transfer via a memory-to-memory move. Recursive transfers and broadcasts (where one value is transferred to the all of the PEs) are similar. Synchronization is needed at transfer points to insure data coherence.

The vision software and simulations were run on a dual-processor Vax 11/780 [5].

3. Overview of the Vision Algorithms

In this section, an overview of the computer vision algorithms is provided. The parameters described are based on the SRI vision module [9] and Fourier descriptors [17].

A simple mechanism for entering an image into the system was desired. In the method chosen, the user employs a terminal with cursor control to draw an image on the screen and enter that image into the data memory. This section of the code used a small subsection of the "curses" [1] utilities available on the test system.

After an image has been entered into the data memory, the first task is to classify the image. This consists of transforming an image comprised of edge and non-edge pixels into an image with edge, internal, and external pixels. An internal pixel is a pixel that represents a point on an object, whereas an external pixel represents a point external to an object (such as the external background or a hole in the object).

After the inside and the outside of the image have been identified by the classification step, the holes in the image are located. A hole is defined as an area outside the object. Thus, the background also fits the definition of a hole. These holes are identified so that later merging can be accomplished easily. This capability is needed since holes that are initially thought to be separate may actually be joined.

The areas of the holes are computed and recorded at the same time as the original hole identification, since the data search patterns are quite similar. For purposes of isolating the object parameters, the background is defined to have an area of zero.

Once the inside of the object is known, the center of mass of the object if determined. Although in and of itself the center of mass is not a particularly useful parameter, it is used to normalize some of the perimeter statistics to be derived later.

To find the perimeter, the edge points that are adjacent to the background are identified. Once this has be done, it is a simple matter to find the distances from the perimeter points to the center of mass. These distances are used to calculate the average, minimum, and maximum perimeter distance from the center of mass.

Finally, using the already determined perimeter, a description of this perimeter is produced in the form of a list of coordinate pairs. This list can then be used to determine Fourier descriptors or other similar parameters. Provisions have been made for the processing of images that contain multiple (non-overlapping) objects.

4. Detailed Description of the Parallel Software

In this section, details of the vision algorithms and of their parallel implementation are presented. Results of the simulation of the parallel algorithms and analysis of the performance of the parallel vision system are presented in Section 5.

Image Initialization

To be able to test the system easily, a simple method by which a user could enter an image into the system was developed. The user executes the vision pro-

gram and then uses a standard keyboard to direct the cursor and draw an image border. The user also has the option of turning the cursor on and off to allow him/her to draw unconnected borders (such as an internal border). The connection pattern for the drawing is an eight neighbor scheme. That is, from a given point, the user can direct the cursor in any of the four horizontal and vertical directions as well as along the diagonals between these directions. After the user has created the image to his/her satisfaction, an exit command automatically starts the image processing on the given image.

The produced image can be saved for later testing and can be reloaded and modified. The user also has the option of either saving the results in a text file or of simply viewing the results as they are produced.

For the parallel implementation, once the image has been created, it is divided among the PEs with each of the PEs having an equally dimensioned stripe (either horizontal or vertical) of the image. Subsequently, each PE operates on the section of the image contained in its local memory, communicating with other PEs when further information is needed.

Internal / External Classification

As a result of the internal/external classification, each pixel is labeled as being on the inside of the object, outside the object, or on the border. The classification scheme implemented is a two-pass method. The first pass traverses the image from the upper left to the lower right. The initial classification of a pixel is based upon the two neighboring points (to the left of the current point and above the current point) that have already been classified. The method tries to classify the new point as external if either of the previous points is external. If the adjacent points are both edges (border pixels), then information about the length of the edge and the previous region classifications are used to make the classification.

The second pass traverses the image from the lower right to the upper left (backwards as compared to the forward pass). This pass uses the four major compass points in relation to the current point to attempt to correct any classification errors. Again, the bias is toward external classification.

This section of the vision software uses several schemes to insure robustness. Besides the ability to reclassify points on the second pass, the software also looks for the specific case of tracing an edge. In addition, several trouble patterns are checked to prevent major misclassifications. Figure 1 illustrates the classification procedure. Figure 1a is the image before classification (border only). The edges are represented by '2.' Figures 1b and 1c are the image after the first and second passes of the classification, respectively. Internal points are represented by '1' and external points are represented by '0.' An example of a reclassification on the second pass is illustrated by the outlined areas in Figures 1b and 1c.

In the parallel implementation, each PE works with its own stripe of the image data. The communication between PEs is limited to the values of the border elements of a subimage. One such transfer will take place for each border element on one of the sides of the subimage. These transfers will be uniform modulo shifts of distance one. As the results show later, this section of the software demonstrates good speedup. Thus, the assumption of a two-pass classifier gives a conservative speedup estimation: if more passes were used, each pass would exhibit the same good speedup.

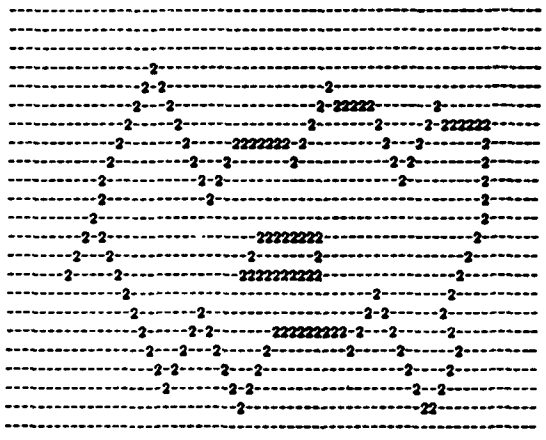


Fig. 1a. Initial image.

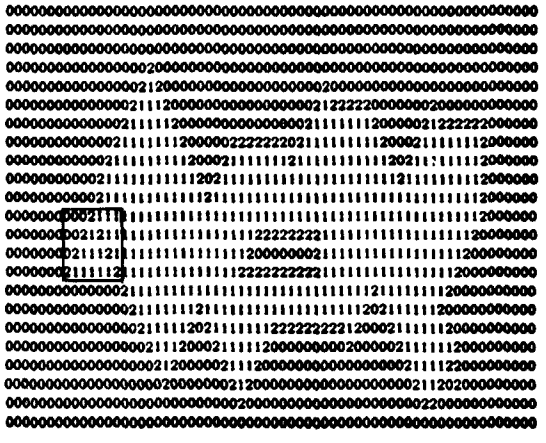


Fig. 1b. Classification: Pass 1.

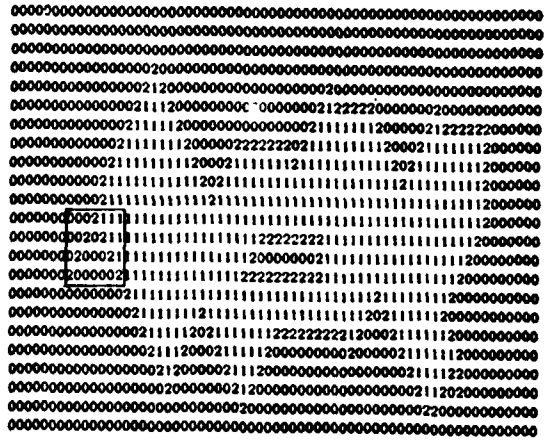


Fig. 1c. Classification: Pass 2.

Identifying Image Holes

After the object has been separated from its surroundings by the classification operation, the holes in the image are identified. This process consists of two steps: initial local hole identification within each PE, followed by merging of holes between PEs. Initial hole labeling is initially performed separately within each PE. This is done by creating a template array in each PE that is of the same size as the subimage in the PE. Each template location contains an identifier that indicates the local hole number for the corresponding subimage point or zero for non-hole points. Each time an external point is located that is not adjacent to a previous hole, a new hole identifier is used and entered for that point in the template. If the external point is adjacent to a previous hole, then the previous identifier is continued. A two-neighbor scheme is used for all of the pixels except those on one of the subimage borders. Since the points on one edge will have only points from the previous row (or column, in the case of horizontal stripes) to base a decision upon, a one-neighbor scheme is used at the borders. The software maintains a set of parameters that keep track of merged holes and their statistics in order to handle the special case of an external point adjacent to two different previous hole identifiers. Experimentation showed that no accuracy problems were encountered due to the small number of neighbors used in the classification.

These operations are performed totally within a PE: no communication with other PEs is needed. Each PE owns the information about its own holes. This information is transferred to other PEs during hole merging (described later). Figure 2 shows the internal hole identifiers for each PE. Hole identifiers that are adjacent (e.g., labels 3, 4, 5, and 6 in PE 2) are considered common. That is, only one of the identifiers contains the information for the hole. All of the others contain a pointer to the "master" information.

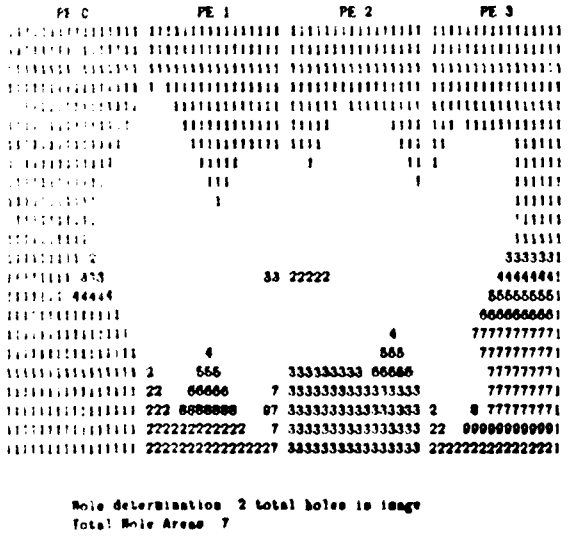


Fig. 2 Image hole determination.

Once the holes have been identified within each PE, they are merged across the PE borders. This is done by transferring the borders of the PE hole template to adjacent processors and searching for matching holes. The areas are merged at the same time that holes are joined. In the scheme used, if a hole has only one edge on a PE border, then the statistics for that hole are transferred to that adjacent PE. This results in each hole being "controlled" by one PE. The information that needs to be transferred from each PE is placed on a transfer stack. These stacks are then transferred. All of these are transfers to logically neighboring PEs (uniform modulo shifts of a distance of one). The amount of information transferred is highly dependent upon the actual image. For purposes of easy identification and to separate holes within an object from the background, the border background is defined as having an area of zero. The process of merging is illustrated in Figure 3.

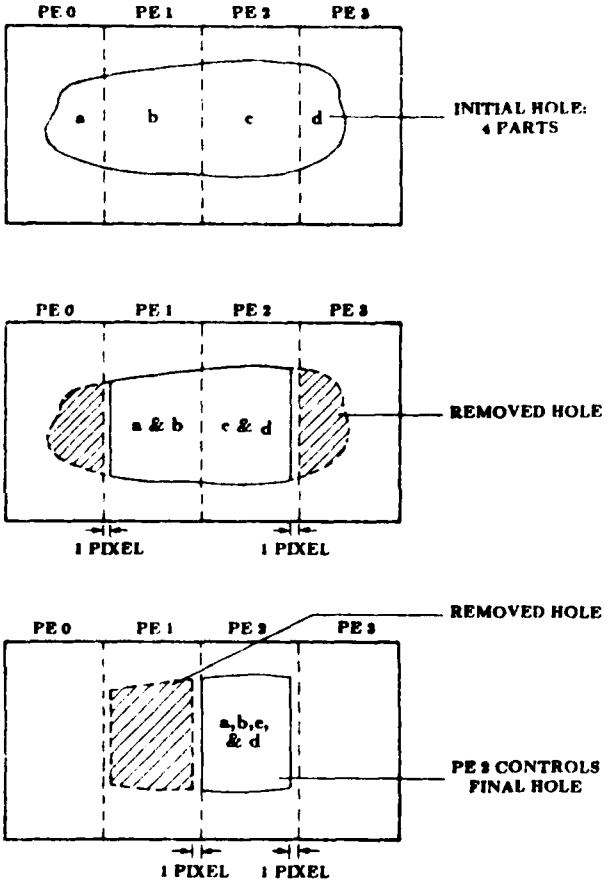


Fig. 3. Hole merging example.

This method of merging holes across PEs is deterministic in that the maximum number of passes needed can be determined by the types of images being examined. For example, the more an object tends to spiral (a spring, for example, as compared to a wheel), the more passes that will be needed. In order to analyze performance, a fixed number of passes (more than necessary for the images considered) was assumed. In simulation, it was found that this section provides poor speedup. Thus, the net result of the fixed large number of passes is again to provide a conservative estimate of the computational speedup of the algorithm.

Computing Image Hole Areas

The areas to be computed are tabulated at the same time as the hole identifiers are placed in the template in each PE. The area computation is therefore divided among the PEs. To handle the merging of holes, either within a PE of between PEs, an indirection table that points to the actual hole area is used.

Locating the Center of Mass

After the points that comprise an object are known, the center of mass of the object can be easily determined. In this system this step is performed by computing the moments in each PE separately and then summing across PEs using recursive doubling [15] (Figure 4). The transfers used are the $cube_i$ functions, $0 \leq i < \log_2 N$. This scheme requires that each PE know its absolute position in the configuration since the weighting of one of the moments in each PE is dependent upon the PE address. For example, if the stripes are in the vertical direction, the the x axis will be split among the PEs. Moments that involve the absolute distance along the x axis will depend upon the PE address. To obtain the center of mass, $\log_2 N$ sets of transfers will be needed. After the center of mass has been determined, it is broadcast to all PEs since this information will be needed at a local PE level in later processing.

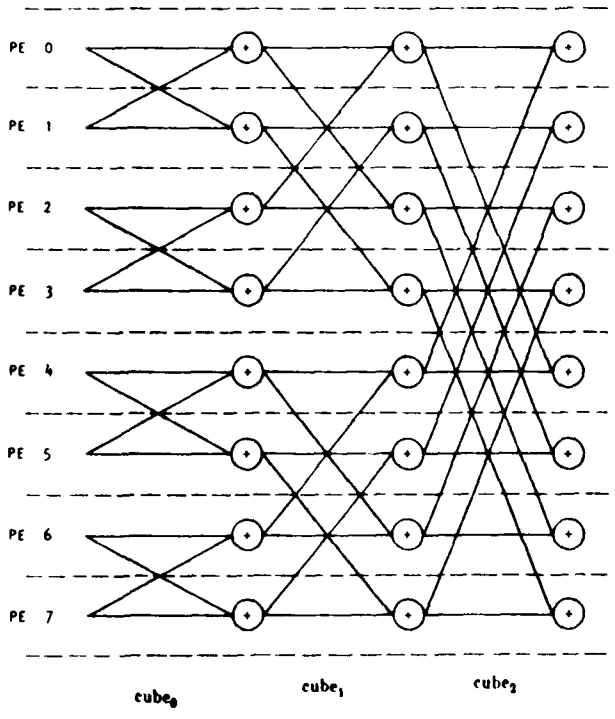


Fig. 4. Example of summing across PEs using recursive doubling.

Perimeter Identification and Perimeter Statistics Determination

Identifying the perimeter is straightforward once the external background hole has been identified. This hole has area zero by definition. An edge point next to an external hole (or next to another perimeter point) is a perimeter point. Since the area of holes is determined through an indirection table, all one needs to do is see if the hole has zero area. When a perimeter point is located in a PE, a counter in that PE is also incremented so that the total perimeter can be determined by a simple application of recursive doubling to accumulate the total across the PEs.

After the perimeter has been identified, it is a simple matter to find the distances between the perimeter points and the previously determined center of mass. This is done by scanning through the image template looking for perimeter points. Each PE scans its stripe of the image. For each perimeter point so found, the radial distance from the perimeter point to the center of mass is determined. A running sum is kept of these distances, along with the minimum and the maximum distances. When the entire image has been scanned, recursive doubling is used to find the average, minimum, and maximum such distances. Three stages of recursive doubling transfers will be needed, one set for each of the perimeter statistics being gathered. This results in a total of $3\log_2 N$ transfers.

Figure 5 shows the identified perimeter for an image. The perimeter is noted by "B," as compared to "2" for a non-perimeter edge point. Figure 6 shows an example of the overall output of the vision software.

Data Preparation for Fourier Descriptors

As an illustration of some of the higher level functions that can be performed once the basic parameters have been extracted, the image can be converted into the information necessary to calculate Fourier descriptors [17]. This information is simply an ordered list representation of the perimeter of the object. Each entry in this list consists of a set of coordinates representing a perimeter point. Fourier descriptors have been proposed as a method of performing shape analysis.

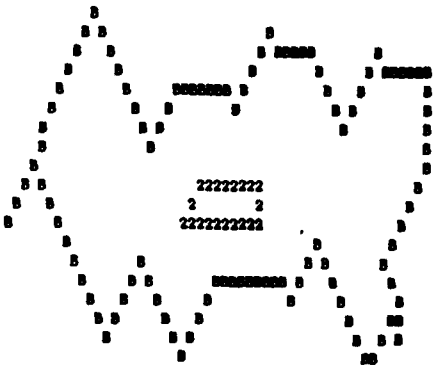
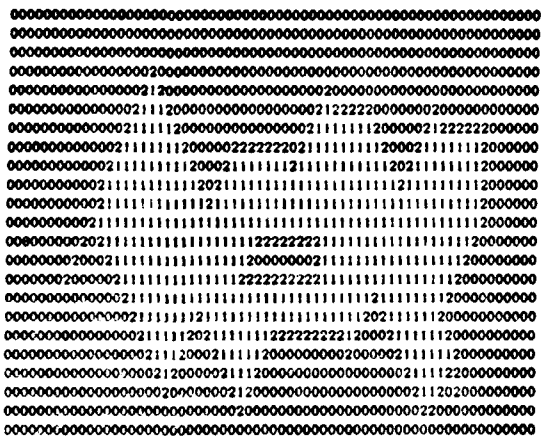


Fig. 5. Object perimeter determination and center of mass statistics.



```
Hole determination  2 total holes in image

Total Object Perimeter  109

Total Hole Areas  7

Center of Mass  (33,11)

Perimeter Statistics  distances from Center of Mass
        MIN = 3,  MAX = 24,  AVG = 12

1 Object(s) in Image
```

Fig. 6. Example of vision software output.

The vision software begins this step by forming the perimeter nodes into a multiply-linked list. This is done to facilitate the removal of false perimeter points (spikes). This converts the perimeter into a traceable contour. Next, these linked-lists are transferred to one PE which completes the processing. This requires uniform modulo shifts of distances from 1 to N - 1. This processing includes converting the lists into partial ordered lists and then combining these lists. Other schemes, such as forming the partial lists in each PE separately, were found to require such a large amount of overhead in transfers that any advantages in parallelism were lost. The final contours in the single PE are then broadcast to the remainder of the PEs in preparation for the Fourier descriptor calculations. If the perimeter is equally distributed among the PEs, (N - 1)/N of the partial ordered listings will need to be transferred. Each of the objects in one of these lists contains ten data fields (two link fields for the linked-list and eight neighbor pointers). If the perimeter is not equally distributed, then the perimeter could be gathered into the PE with the largest number of perimeter points, and this will require fewer total transfers. Thus, if there are P perimeter points, a maximum of (N - 1)P/N transfers will be needed.

Multiple Object Images

The software that has been described to this point has treated the contents of the image field as one object. If there is more than one object in the image field, the same software can still be used, but the results will be a composite of the information for the separate objects. However, it is not exceedingly difficult to separate the information for the separate objects.

Once the contours of the image have been determined, the software knows how many separate objects are in the image. This involves the classification, hole and area identification and merging, and perimeter determination steps described above. That is, the number of contours will equal the number of objects in the image given that the objects do not overlap and that no object is inside another (such as a bolt within a wheel rim). The items can be processed individually by removing the objects corresponding to the undesired contours and reprocessing the image. This can be done for each object in the image. The individual processing involves all the the previous steps, from classification through perimeter determination and perimeter statistics.

To remove an object from the image, its perimeter points (which are known from the contour) are marked to be removed. Two passes are made over the image (similar to the initial classification) to convert internal, perimeter, and edge points bordering the removal points to removal points themselves. This is similar to the erosion scheme used by CLIP4 [11]. A final pass is made over the image to convert all removal points to external points, effectively erasing the object from the image.

If the program detects multiple images, it will still give the composite results, but it will also sequentially erase all but one of the objects and then process the remaining object. This additional processing is identical to the main processing sequence, except the checks for multiple objects are omitted.

Additional Parameters

Other parameters may be added to a vision system in order to improve the robustness of object identification. Some of these additional parameters are simply combinations of previous parameters. An example of such a parameter is the factor of roundness (how circular the image is), which is computed by dividing 4π times the area by the square of the perimeter. The area of the object could also be calculated at the same time that the second classification pass is made. This area could be combined with the internal hole area to provide a total of the areas occupied by the object. The ratio of hole area to total area is similarly obtainable.

There are other parameters that would require additional computation in the main processing sequence. This class of parameters would include such features as second moments, ratios of major and minor axes, finding the bounding rectangle, and line fitting. Others could be added based upon the specific task at hand.

Finally, one needs to consider the non-ideal cases where either multiple objects in the image overlap or the objects are not entirely contained within the borders. Much information for the latter case can be obtained from processing the object as usual and then applying statistical methods to determine possible matches with known objects. The other case is not as simple - some type of image reduction would be necessary if it was determined that an object was not known. Such software could selectively reduce protrusions of an object until a known object was found.

5. Analysis

In order to evaluate the use of the parallel architecture for computer vision, analytical comparisons of the parallel and serial algorithms were performed and the simulation of the parallel software was compared to the serial implementation. An estimation of the computational speedups was derived by an examination of the structure of the parallel software. Table 1 summarizes the speedups for the major algorithms. The proportions

Table 1
Computational Performance Results

Algorithm Division	Approx. Speedup	Serial Time	Time Proportions
class()	N(I/(I+N-1))	15.36	0.3531
holes()	N/((N-1)(SPIFAC+1))	15.79	0.3630
areas()	(called by holes)	N/A	N/A
center()	N	1.64	0.0377
pstats()	N	10.71	0.2462

I = Image Border (I by I image) N = Number of PEs

SPIFAC = How many times a section of the object in the image can switch directions in crossing the image (for example, the letter "Z" would have a SPIFAC of 2). For the images analyzed, SPIFAC = 6.

of time required by different sections of the code were determined by executing a serial version of the algorithm. The time proportions are used to provide a weighting of the parallel speedup results. In this way, a section with low speedup that requires only a small fraction of the serial processing time will not falsely lower the overall speedup. Similarly, a section with high speedup that requires only a small fraction of the serial processing time will not falsely raise the overall speedup. Using the time proprotions, the total weighted speedup *S(N)* can be computed:

$$S(N) = \frac{0.3531NI}{I+N-1} + \frac{0.3630N}{(N-1)(SPIFAC+1)} + 0.0377N + 0.2462N$$

$$= N\left\{\frac{0.3531I}{I+N-1} + \frac{0.3630}{(N-1)(SPIFAC+1)} + 0.2839\right\}$$

One measure of the performance of a parallel algorithm is the *efficiency E(N)*, defined to be the ratio of the speedup to the number of processors [8]. Table 2 shows the speedup and efficiency for the case of a 64 by 64 image. For the example, although the speedup increases with N, the rate of increase is not proportional to N and the efficiency decreases gradually with N.

The experimental results for the major sections of the software are presented in Table 3. The simulations were designed to provide a conservative estimate of the speedup; assumptions about transfer timings and synchronization delays could only be approximated. The problem of non-determinism in speedups was handled by using deterministic versions of non-deterministic routines. Again, these routines were designed to provide a conservative estimate of the speedup. No overlap of processing and transfers was assumed, although in many situations, inter-PE transfers can be performed at the same time

Table 2
Speedup and Efficiency for I=64

N	S(N)	E(N)
2	1.37	0.683
4	2.55	0.638
8	4.88	0.610
16	9.17	0.573
32	16.7	0.523
64	29.6	0.463

To be compatible with the "curses" input method, images were 64 by 23. The image was divided into 64/N by 23 stripes.

that independent processing is occurring. The simulation results can therefore be used as a rough indicator of the speedup obtained by the parallel algorithms. Both the analytic and experimental results bear out the observation that the speedup will not grow as N, because the algorithms in which the largest proportion of time is spent (classification and hole location) have less than ideal speedup. (The experimental speedups are somewhat less than the analytic speedups due to the conservative assumptions made throughout the simulation and the non-square image used.) Simulation demonstrated that the major problem with the parallel implementation is

Table 3
Experimental Speedup Results

Algorithm	Serial Time	N=2 Time	N=4 Time	N=2 Speedup	N=4 Speedup
classification	15.36	9.47	6.02	1.62	2.55
holes and areas	15.79	13.47	17.11	1.17	0.92
center	1.64	1.11	0.66	1.48	2.48
perimeter	10.71	5.61	2.79	1.91	3.84
overall	43.50	29.64	26.86	1.47	1.62

basically of one form: the number of transfers needed reduces the effectiveness of the parallelism. This can occur when the amount of information that is needed to make a proper decision (such as for hole merging) is large. This problem can manifest itself in several forms, such as algorithms that are inherently serial or that require data from the entire image. Such tasks might better be performed in one PE or in the control unit.

6. Architectural Considerations

A specific type of architecture has been assumed throughout this simulation and analysis. At this point, this restriction will be removed and the tasks considered will be examined to explore a parallel architecture tailored to the characteristics of the vision task.

By examining the algorithms, is it seen that a given memory area (the memory assigned to one PE) is not needed by more than two PEs in a given processing section If the memory is dual ported, with one write channel and two read channels, then the need for transfers can be virtually eliminated. In such an approach, the memory that was previously the exclusive responsibility of a specific PE would still be connected to that PE via the write channel and one of the read channels. However, the other read channel would be connected to a memory redirection network that would be setable by the control unit when a new type of access pattern is needed. This redirection network could either be bidirectional or (more practical) two unidirectional networks, one direction being used to transmit the memory addresses and the other being used to return the data. The advantage of using two unidirectional networks is that information can be flowing in both directions at the same time without the need for redirection or buffering. This would allow the memory to be accessed in an interleaved manner, further improving system performance. When this scheme is compared with the number of transfers needed in some of the processing steps (such as in hole merging and Fourier descriptor preparation), the possible savings are quite evident.

7. Summary

In this paper, analytic and simulation results for the application of parallel processing to the computer vision task have been presented. In general, it has been shown that for moderate numbers of processors, increases in performance (such as overall speedup) on the order of I for an I by I image are obtainable. Because of the modular design of the software developed, it is quite possible to expand the processing sequence to include other common image processing techniques. From the analytic and simulation capabilities described, given specific speed requirements for a particular vision task and assumptions about processor speed, it will be possible to determine the number of processors needed to satisfy the task requirements. This work contributes to the understanding of the design of parallel systems for image processing applications.

References

[1] K. Arnold, "Screen Updating and Cursor Movement Optimization, A Library Package," Unix• Version 4 1bsd.

[2] K. E. Batcher, "The Design of a Massively Parallel Processor," *IEEE Trans. Comp.*, Vol. C-29, Sept. 1980, pp. 836-844.

[3] M. J. B. Duff, "Parallel Algorithms and Their Influence on the Specification of Application Problems," in *Multicomputers and Image Processing: Algorithms and Programs*, K. Preston and L. Uhr, eds., Academic Press, New York, NY, 1982.

[4] M. J. Flynn, "Very High-Speed Computing Systems," *Proc. IEEE*, Vol. 54, Dec. 1966, pp. 1901-1909.

[5] G. H. Goble and M. H. Marsh, "A Dual Processor Vax•• 11/780," *IEEE 9th Annual Symp. on Comp. Arch.*, Apr. 1982, pp. 291-298.

[6] A. Gottlieb et al., "The NYU Ultracomputer -- Designing an MIMD Shared Memory Parallel Computer," *IEEE Trans. Comp.*, Vol. C-32, Feb. 1983, pp. 175-189.

[7] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice Hall, Inc., Englewood Cliffs, NJ, 1978.

[8] D. J. Kuck, "A Survey of Parallel Machine Organization and Programming," *Computing Surveys*, Vol. 9, Mar. 1977, pp. 29-59.

[9] D. Nitzan et al., "Machine Intelligence Research Applied to Industrial Automation," SRI Report, Menlo Park, CA 94025, Aug. 1979.

[10] M. C. Pease, 'The Indirect Binary n-cube Microprocessor Array," *IEEE Trans. Comp.*, Vol. C-26, May 1977, pp. 458-473.

[11] D. E. Reynolds and G. P. Otto, "Software Tools for CLIP4," Report No. 82/1, Dept. of Physics and Astronomy, University College, London, Jan. 1981.

[12] H. J. Siegel, "Analysis Techniques for SIMD Machine Interconnection Networks and the Effects of Processor Address Masks," *IEEE Trans. Comp.*, Vol. C-26, Feb. 1977, pp. 153-161.

[13] H. J. Siegel, *Interconnection Networks for Large Scale Parallel Processing: Theory and Case Studies*, D. C. Heath and Co., Lexington, MA, 1983.

[14] H. J. Siegel et al., "PASM: A Partitionable SIMD/MIMD System for Image Processing and Pattern Recognition," *IEEE Trans. Comp.*, Vol. C-30, Dec. 1981, pp. 934-947.

[15] H. S. Stone, ed., *Introduction to Computer Architecture*, Science Research Associates, Chicago, IL, 1980, pp. 394-396.

[16] R. J. Swan et al., "The Implementation of the Cm• Multi-microprocessor," *AFIPS 1977 Nat'l. Comp. Conf.*, June 1977, pp. 645-655.

[17] T. P. Wallace and O. R. Mitchell, "Analysis of Three-Dimensional Movement Using Fourier Descriptors," *IEEE Trans. Pattern Analysis and Machine Intelligence*, Vol. PAMI-2, Nov. 1980, pp. 583-588.

[18] W. Wulf and C. Bell, "C.mmp -- A Multi-miniprocessor," *AFIPS 1972 Fall Joint Comp. Conf.*, Dec. 1972, pp. 765-777

• Unix is a trademark of Bell Laboratories.

••VAX is a trademark of Digital Equipment Corporation.

Paper 2

Parallel Processing for Computer Vision

From *Integrated Technology for Parallel Image Processing*, S. Levialdi, editor, Academic Press, 1985.

Chapter Five

*Parallel Processing for Computer Vision**

Thomas A. Rice and Leah H. Jamieson

1. INTRODUCTION

Parallel processing has the potential of providing fast, flexible solutions to many computationally intensive tasks. In this paper, the use of parallelism for computer vision is described. Considerations for the design of a parallel architecture for computer vision are discussed.

The vision task consists of a number of different algorithms; several of the algorithms have markedly different computational characteristics. It is possible to achieve real-time implementations of some sequences of vision algorithms in hardware. The use of parallel processing allows significantly greater flexibility, both in the types of images that can be processed (e.g., gray-level images as well as binary) and in the choice of vision algorithms used. The work here presents theoretical analyses and simulation results for a collection of individual algorithms and for the overall vision task. This paper extends the work reported in Rice and Siegel [1].

2. DEFINITIONS FOR THE PARALLEL SIMULATION

In this section, two general models of parallel computation are defined, and the specific model used for the computer vision task is presented. The implementation of the parallel simulation is described.

* This research was supported by the United States Army Research Office, Department of the Army, under grant number DAAG29-82-K-0101.

INTEGRATED TECHNOLOGY FOR PARALLEL IMAGE PROCESSING
ISBN 0-12-444820-8

2.1 Model

Single instruction stream-multiple data stream (SIMD) machines [2] represent a form of synchronous, highly parallel processing. Systems with up to 1 000 full processors have been proposed [3], [4]; systems with as many as 9 000 and 16 000 simple processors have been built [5], [6]. An SIMD machine typically consists of a *control unit*, a set of P *processing elements (PEs)*, each a processor with its own memory, and an *interconnection network*. The control unit broadcasts instructions to all PEs, and each active PE executes the instruction on the data in its own memory. The interconnection network allows data to be transferred among the PEs. SIMD machines are especially well-suited for exploiting the parallelism inherent in certain tasks performed on vectors and arrays.

Multiple instruction stream–multiple data stream (MIMD) machines [2] represent asynchronous parallel processing. MIMD systems with 16 [7] and 50 [8] processors have been built; MIMD systems with as many as 4 000 processors [9] have been proposed. An MIMD machine typically consists of P processors and M memories, M ≥ P, where each processor can follow an independent instruction stream. As with SIMD machines, there is a multiple data stream and an interconnection network. Thus, there are P independent processors that can communicate among themselves. There may be a coordinator unit to oversee the activities of the processors.

The parallel machine model assumed for the computer vision task consists of a set of PEs under the management of a control unit. The number of PEs is a power of two: $N = 2^n$. Each of the PEs has a unique address between 0 and $N - 1$. In addition, there exists an interconnection network to allow the simultaneous transfer of data among the PEs. For the computer vision task, the transfer patterns required will be uniform modulo shifts and cube interconnection functions. In a *uniform modulo shift*, PE j transfers data to PE $(j + d)$ modulo N for all j, $0 \leq j < N$, given a positive or negative integer distance d. The value of d may vary from one transfer to the next; however, for a given transfer, all PEs will send their data the same distance d. The set of *cube interconnection functions* consists of $n = \log_2 N$ functions, cube_i, for $0 \leq i < n$ [10]. If $P_{n-1} \cdots P_i \cdots P_0$ is the binary representation of a PE's address, then the cube_i function exchanges data between all pairs of PEs whose addresses differ in bit i:

$$\text{cube}_i(P_{n-1} \cdots P_i \cdots P_0) = P_{n-1} \cdots \bar{P}_i \cdots P_0$$

The model assumed here combines SIMD and MIMD attributes. Each PE contains the same code but executes the code on a different subimage. However, within each PE, the code can run in MIMD mode. This

modification to the basic models allows faster execution on some code than a pure SIMD model, without incurring the expense of the full flexibility of an MIMD machine. The gains in speed will occur on the execution of conditional statements:

where < condition > do < block 1 >
elsewhere do < block 2 >

In SIMD mode, those PEs satisfying the < condition > execute < block 1 >. Then the remaining PEs execute < block 2 >. In the model here, < block 1 > and < block 2 > will be executed concurrently but in different sets of PEs. On the other hand, this is not full MIMD mode, as it is required that the code in each PE be the same. This aids in enforcing data coherence, e.g., insuring that a PE acquires the correct version of a variable from another PE.

Synchronization can take place in one of two ways. First, synchronization is required at all data transfer points, because data transfers often involve the same variable for all of the PEs. Even if the separate processors take different times to execute their code, they will be forced to synchronize at transfers to insure coherence. Explicit synchronization is also possible by one of the simulation language constructs that requires that all PEs finish a section of code before any can move to the next section of code.

The motivation for the assumed model comes from two directions. First, for many image processing operations, it is natural to consider executing the same code on subimages of the original image. Each subimage is a valid image, and the same types of operations are needed on the pixels of each subimage. Second, since the actual quantities of the various operations that will be performed on each subimage may vary, asynchronous operation may allow higher PE utilization.

This hybrid mode of operation may not be suitable for some algorithms. The requirements for such a mode to be useful are (1) that the PEs contain and execute the same code, with possible differences based only on the evaluation of conditional statements, and (2) that the need to synchronize at data transfers does not cancel the gains obtained by simultaneous evaluation of conditionals. For the vision algorithms examined here, these requirements are met.

2.2 Simulation

There are two major approaches to the development of parallel software. Either the software can be of a generally descriptive nature to illustrate

the parallelism (or lack thereof) inherent in a task, or the software can be designed to be compilable and testable, either by parallel execution or serial simulation. Due to the computational intensity and intricacy of the computer vision task, the most reliable way to insure correctness is by testing. This guarantees that typical problem cases are being handled correctly by testing the software for a variety of images. A set of test images, some with multiple objects, was used for debugging and for analyzing computational speedup. Therefore, the software was designed so that it could be compiled and tested.

Programming was done in a modified version of the C language [11]. This language was chosen for the capabilities it provides for developing parallel data structures and the high degree to which one can manipulate system information (such as memory areas). The latter played a large part in the simulating of parallel data transfers. The actual conversion of the serial C language to a parallel language was done by means of macros and support subroutines. These features were designed to facilitate the development of parallel code without requiring the user to know the specific details of the serial implementation. Thus, one can simply use the macro file without knowing its details and can then write parallel code.

The major points of this implementation are as follows. A construct of the form

```
in_pe { codeblock; }
```

executes the enclosed block of code in each of the PEs. The prefix "PE." prepended to a variable indicates that the variable is local to a PE. All other variables are assumed to be global (i.e., the control unit has one copy of the variable). Global variables are used for such operations as loop control and overall conditional testing. There are also versions of the "in_pe" construct that allow the code to be executed in a limited subset of the PEs. These schemes use an address mask [12], which is a matching format that the PE address must match for execution to occur in that PE.

Interprocessor communication is accomplished by a *transfer* subroutine:

```
transfer (destination_address, source_address, offset)
```

The transfer routine uses these addresses along with information about the size and structure of the PE data space to simulate the transfer by a memory-to-memory move. Recursive transfers and broadcasts (in which one value is transferred to all of the PEs) are similar. Synchronization is needed at transfer points to insure data coherence.

The vision software and simulations were run on a dual-processor Vax 11/780 [13].

3. OVERVIEW OF THE VISION ALGORITHMS

In this section, an overview of the computer vision algorithms is provided. The parameters described are based on the SRI vision module [14] and Fourier descriptors [15].

A simple mechanism for entering an image into the system was desired. In the method chosen, the user employs a terminal with cursor control to draw an image on the screen and enter that image into the data memory. This section of the code used a small subsection of the "curses" [16] utilities available on the test system. This was later expanded to allow other image formats to be input. The images used here and in the subsequent steps are assumed to be binary images, although the algorithms can be generalized to handle gray-level images.

After an image has been entered into the data memory, the first task is to classify the image. This consists of transforming an image comprised of edge and non-edge pixels into an image with edge, internal, and external pixels. An internal pixel is a pixel that represents a point on an object, whereas an external pixel represents a point external to an object (such as the external background or a hole in the object).

After the inside and the outside of the image have been identified by the classification step, the holes in the image are located. A hole is defined as an area outside the object. Thus, the background also fits the definition of a hole. These holes are identified so that later merging can be accomplished easily. This capability is needed because holes that are initially thought to be separate may actually be joined.

The areas of the holes are computed and recorded at the same time as the original hole identification, because the data search patterns are similar. For purposes of isolating the object parameters, the background is defined to have an area of zero.

Once the inside of the object is known, the center of mass of the object is determined. Although in and of itself the center of mass is not a particularly useful parameter, it is used to normalize some of the perimeter statistics to be derived later.

To find the perimeter, the edge points that are adjacent to the background are identified. Once this has been done, it is a simple matter to find the distances from the perimeter points to the center of mass. These distances are used to calculate the average, minimum, and maximum perimeter distance from the center of mass.

Finally, using the already determined perimeter, a description of this perimeter is produced in the form of a list of coordinate pairs. This list can then be used to determine Fourier descriptors or other similar parameters.

Provisions have been made for the processing of images that contain multiple (nonoverlapping) objects.

4. DETAILED DESCRIPTION OF THE PARALLEL SOFTWARE

In this section, details of the vision algorithms and of their parallel implementation are presented. Results of the simulation of the parallel algorithms and analysis of the perfomance of the parallel vision system are presented in Section 5.

4.1 Image initialization

To be able to test the system easily, a simple method by which a user could enter an image into the system was developed. The user executes the vision program and then uses a standard keyboard to direct the cursor and draw an image border. The user also has the option of turning the cursor on and off to allow the drawing of unconnected borders (such as an internal border). The connection pattern for the drawing is an eight-neighbor scheme. That is, from a given point, the user can direct the cursor in any of the four horizontal and vertical directions as well as along the diagonals between these directions.

The screen size does not limit the size of the image being created, as the screen merely acts as a window into the image. During image creation the current position of the cursor is maintained in the upper left-hand corner of the screen. Messages and inputs are handled on the lowest line of the screen. If the drawing gets too near to any of the borders, the window into the image is automatically moved. The user can also specify a location to which to move the cursor. If this position is not in the current window, the window is automatically moved. All borders are strictly enforced: The user cannot draw beyond the edge of the border under any condition. After the user has created the image, an exit command automatically starts the image processing on the image.

In addition, images with 256 gray levels that are stored as character arrays (e.g., one character per pixel) can be loaded by the system. Simple thresholding routines as well as a Sobel operator are automatically applied to such images to convert them into binary images. The user is prompted for the thresholds for each image.

The produced image can be saved for later testing and can be reloaded and modified. The user also has the option of saving the results in a text file or of viewing the results as they are produced.

For the parallel implementation, once the image has been created, it is divided among the PEs with each of the PEs having an equally dimensioned stripe (either horizontal or vertical) of the image. Subsequently, each PE operates on the section of the image contained in its local memory, communicating with other PEs when further information is needed.

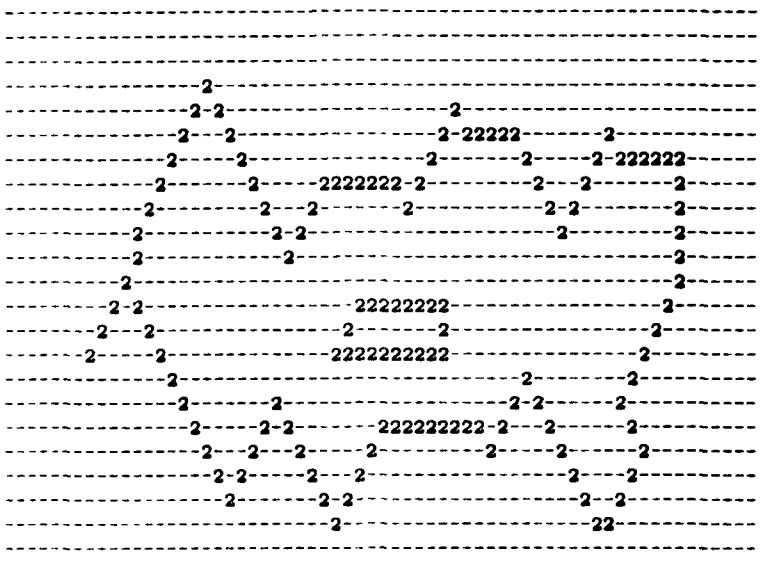
4.2 Internal/external classification

The internal/external classification labels each pixel as being on the inside of the object, outside the object, or on the border. The classification scheme implemented is a two-pass method. The first pass traverses the image from the upper left to the lower right. The initial classification of a pixel is based on the two neighboring points (to the left of the current point and above the current point) that have already been classified. The method tries to classify the new point as external if either of the previous points is external. If the adjacent points are both edges (border pixels), then information about the length of the edge and the previous region classifications are used to make the classification.

The second pass traverses the image from the lower right to the upper left (backward, as compared with the forward pass). This pass uses the four major compass points in relation to the current point to attempt to correct any classification errors. Again, the bias is toward external classification.

This section of the vision software uses several schemes to insure robustness. Besides the ability to reclassify points on the second pass, the software also looks for the specific case of tracing an edge. In addition, several trouble patterns are checked to prevent major misclassifications. Figure 1 illustrates the classification procedure. Figure 1(a) is the image before classification (border only). The edges are represented by '2.' Figures 1(b) and 1(c) are the image after the first and second passes of the classification, respectively. Internal points are represented by '1,' and external points are represented by '0.' An example of a reclassification on the second pass is illustrated by the outlined areas in Fig. 1(b) and 1(c).

In the parallel implementation, each PE works with its own stripe of the image data. The communication between PEs is limited to the values of the border elements of a subimage. One such transfer takes place for each border element on one of the sides of the subimage. These transfers are uniform modulo shifts of distance one. As the results show later, this section of the software demonstrates good speedup. Thus, the assumption of a two-pass classifier gives a conservative speedup estimation: if more passes were used, each pass would exhibit the same good speedup.

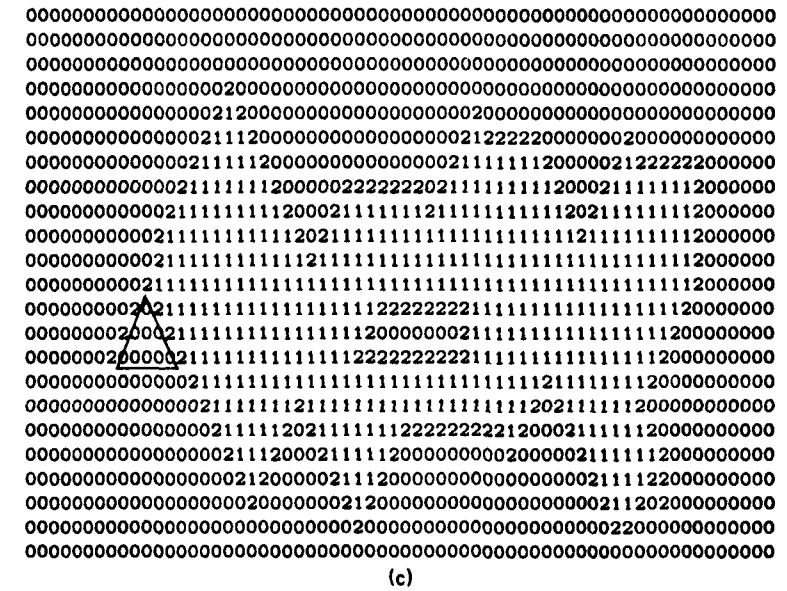


(a)

Fig. 1(a) Initial image

(b)

Fig. 1(b) Classification: Pass 1



(c)

Fig. 1(c) Classification: Pass 2

4.3 Identifying image holes

After the object has been separated from its surroundings by the classification operation, the holes in the image are identified. This process consists of two steps: initial local hole identification within each PE, followed by merging of holes between PEs. Initial hole labeling is first performed separately within each PE. This is done by creating a template array in each PE that is of the same size as the subimage in the PE. Each template location contains an identifier that indicates the local hole number for the corresponding subimage point, or zero for non-hole points. Each time an external point is located that is not adjacent to a previous hole, a new hole identifier is used and entered for that point in the template. If the external point is adjacent to a previous hole, then the previous identifier is continued. A two-neighbor scheme is used for all the pixels except those on one of the subimage borders. Since the points on one edge have only points from the previous row (or column, in the case of horizontal stripes) upon which to base a decision, a one-neighbor scheme is used at the borders. The software maintains a set of parameters that keeps track of merged holes and their statistics in order to handle the special case of an external point adjacent to two different previous

hold identifiers. Experimentation showed that no accuracy problems were introduced by the small number of neighbors used in the classification.

These operations are performed totally within a PE: no communication with other PEs is needed. Each PE owns the information about its own holes. This information is transferred to other PEs during hole merging (described later). Figure 2 shows the internal hole identifiers for each PE. Hole identifiers that are adjacent (e.g., labels 3, 4, 5, and 6 in PE 2) are considered common. That is, only one of the identifiers contains the information for the hole. All of the others contain a pointer to the "master" information.

Once the holes have been identified in each PE, they are merged across the PE borders. This is done by transferring the borders of the PE hole template to adjacent processors and searching for matching holes. The areas are merged at the same time that holes are joined. In the scheme used, if a hole has only one edge on a PE border, then the statistics for that hole are transferred to that adjacent PE. This results in each hole being "controlled" by one PE. The information that needs to be transferred from each PE is placed on a transfer stack. These stacks are then transferred. All of these are transfers to logically neighboring PEs (uniform modulo shifts of a distance of one). The amount of information transferred is highly dependent on the

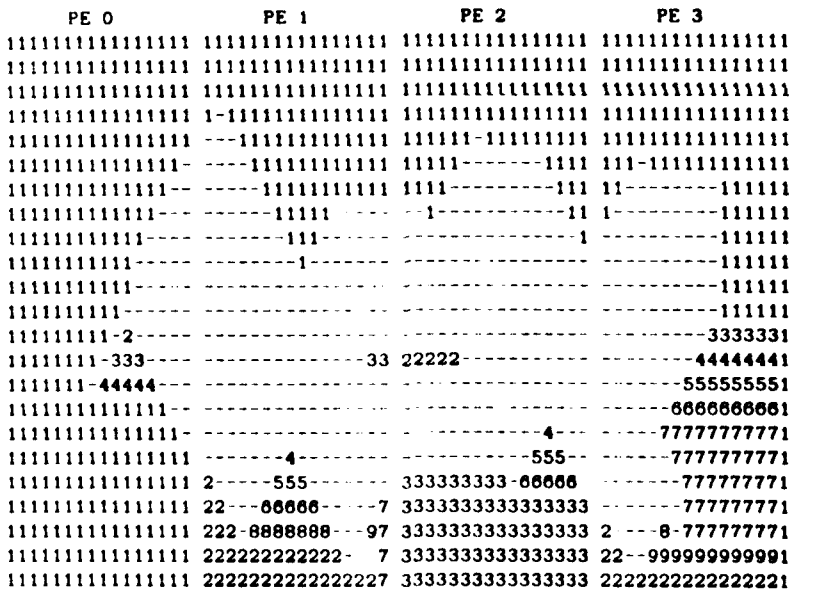


Fig. 2 Image hole determination

actual image. For purposes of easy identification and for separation of holes within an object from the background, the border background is defined as having an area of zero. The process of merging is illustrated in Fig. 3.

This method of merging holes across PEs is deterministic in that the maximum number of passes needed can be determined by the types of images being examined. For example, the more an object tends to spiral (a spring, for example, as compared with a wheel), the more passes are needed. To

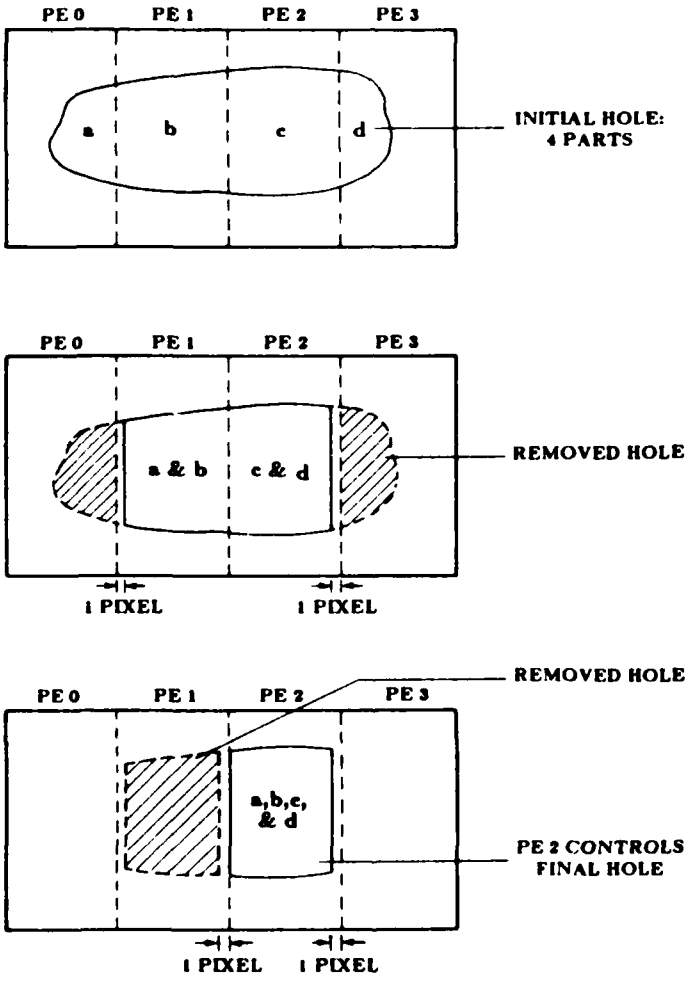


Fig. 3 Hole merging example

analyze performance, preliminary tests assumed a fixed number of passes (more than necessary for the images considered). In simulation, it was found that this section provides poor speedup. Thus, for this step, the net result of the fixed large number of passes is again a conservative estimate of the computational speedup of the algorithm. A refinement of the algorithm was also tested. By using only the required number of passes, appreciable improvements in speedup were obtained.

4.4 Computing image hole areas

The areas to be computed are tabulated at the same time as the hole identifiers are placed in the template in each PE. The area computation is therefore divided among the PEs. To handle the merging of holes, either within a PE or between PEs, an indirection table that points to the actual hole area is used.

4.5 Locating the center of mass

After the points that comprise an object are known, the center of mass of the object can be easily determined. In this system this step is performed by computing the moments in each PE separately and then summing across PEs using recursive doubling [17] (Fig. 4). The transfers used are the cube$_i$ functions, $0 \leq i < \log_2 N$. This scheme requires that each PE know its absolute position in the configuration because the weighting of one of the moments in each PE is dependent upon the PE address. For example, if the stripes are in the vertical direction, then the x axis is split among the PEs. Moments that involve the absolute distance along the x axis depend on the PE address. To obtain the center of mass, $\log_2 N$ sets of transfers are needed. After the center of mass has been determined, it is broadcast to all PEs, because this information is needed at a local PE level in later processing.

4.6 Perimeter identification and perimeter statistics determination

Identifying the perimeter is straightforward once the external background hole has been identified. This hole has area zero by definition. An edge point next to an external hole (or next to another perimeter point) is a perimeter point. Since the area of holes is determined through an indirection table, all one needs to do is see if the hole has zero area. When a perimeter point is

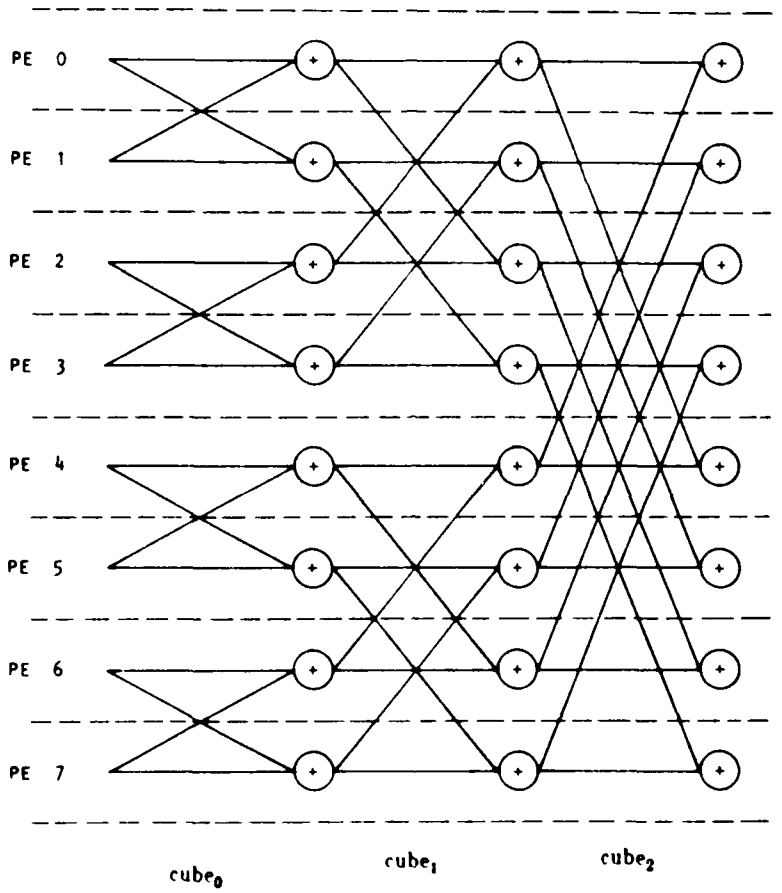


Fig. 4 Example of summing across PEs using recursive doubling

located in a PE, a counter in that PE is also incremented so that the total perimeter can be determined by a simple application of recursive doubling to accumulate the total across the PEs.

After the perimeter has been identified, it is a simple matter to find the distances between the perimeter points and the previously determined center of mass. This is done by scanning through the image template looking for perimeter points. Each PE scans its stripe of the image. For each perimeter point found, the radial distance from the perimeter point to the center of mass is determined. A running sum is kept of these distances, along with the minimum and the maximum distances. When the entire image has been

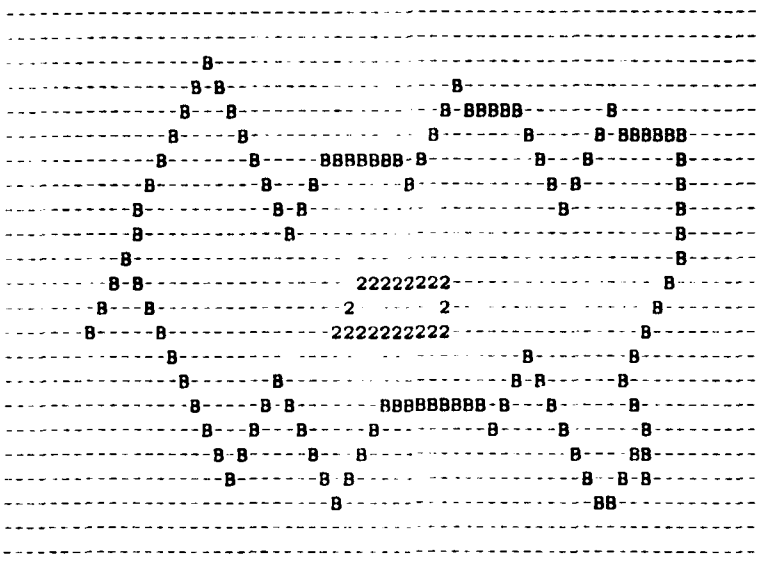


Fig. 5 Object perimeter determination and center-of-mass statistics. Total object perimeter is 109; the centre of mass, (33, 11); distances from center of mass to perimeter, min 3, max 24, average 12

Fig. 6 Example of vision software output. Two holes in image; total perimeter is 109; total hole area 7; center of mass (33, 11); distances from center of mass to perimeter, min 3, max 24, average 12; one object in image

scanned, recursive doubling is used to find the average, minimum, and maximum distances. Three stages of recursive doubling transfers are needed, one set for each of the perimeter statistics being gathered. This results in a total of 3 $\log_2 N$ transfers.

Figure 5 shows the identified perimeter for an image. The perimeter (border) is noted by "B," as compared with "2" for a nonperimeter edge point. Figure 6 shows an example of the overall output of the vision software.

4.7 Data preparation for Fourier descriptors

As an illustration of some of the higher level functions that can be performed once the basic parameters have been extracted, the image can be converted into the information necessary to calculate Fourier descriptors [15]. This information is simply an ordered list representation of the perimeter of the object. Each entry in this list consists of a set of coordinates representing a perimeter point. Fourier descriptors have been proposed as a method of performing shape analysis.

The vision software begins this step by forming the perimeter nodes into a multiply linked list, which facilitates the removal of false perimeter points (spikes). This converts the perimeter into a traceable contour. Next, these linked lists are transferred to one PE which completes the processing. This requires uniform modulo shifts of distances from 1 to $N - 1$. This processing includes converting the lists into partial ordered lists and then combining these lists. Other schemes, such as forming the partial lists in each PE separately, were found to induce such a large amount of overhead in transfers that any advantages in parallelism were lost. The final contours in the single PE are then broadcast to the remainder of the PEs in preparation for the Fourier descriptor calculations. If the perimeter is equally distributed among the PEs, $(N - 1)/N$ of the partial ordered listings need to be transferred. Each of the objects in one of these lists contains ten data fields (two link fields for the linked list and eight neighbor pointers) If the perimeter is not equally distributed, then the perimeter could be gathered into the PE with the largest number of perimeter points, and this requires fewer total transfers. Thus, if there are P perimeter points, a maximum of $(N - 1)\ P/N$ transfers are needed.

4.8 Multiple object images

The software that has been described has treated the content of the image field as one object. If there is more than one (nonoverlapping) object in the image field, the same software can still be used, but the results will be a

composite of the information for the separate objects. However, it is not exceedingly difficult to separate the information for the separate objects.

Once the contours of the image have been determined, the software knows how many separate objects are in the image. This involves the classification, hole and area identification and merging, and perimeter determination steps described above. That is, the number of contours equals the number of objects in the image, given that the objects do not overlap and that no object is inside another (such as a bolt in a wheel rim). The items can be processed individually by removing the objects corresponding to the undesired contours and reprocessing the image. This can be done for each object in the image. The individual processing involves all the the previous steps, from classification through perimeter determination and perimeter statistics.

To remove an object from the image, its perimeter points (known from the contour) are marked to be removed. Two passes are made over the image (similar to the initial classification) to convert internal, perimeter, and edge points bordering the removal points to removal points themselves. This is similar to the erosion scheme used by CLIP4 [18]. A final pass is made over the image to convert all removal points to external points, effectively erasing the object from the image.

If the program detects multiple images, it still gives the composite results, but it also sequentially erases all but one of the objects and then processes the remaining object. This additional processing is identical to the main processing sequence, except that the checks for multiple objects are omitted.

4.9 Additional parameters

Other parameters can be added to a vision system to improve the robustness of object identification. Some of these additional parameters are simply combinations of previous parameters. An example of such a parameter is the factor of roundness (how circular the image is), which is computed by dividing 4π times the area by the square of the perimeter. The area of the object could also be calculated at the same time that the second pass in the internal/external classification step is made. This area could be combined with the internal hole area to provide a total of the areas occupied by the object. The ratio of hole area to total area is similarly obtainable.

There are other parameters that would require additional computation in the main processing sequence. This class of parameters includes such features as second moments, ratios of major and minor axes, finding the bounding rectangle, and line fitting. Others could be added, based upon the specific task at hand.

Finally, one needs to consider the nonideal cases in which multiple objects in the image overlap or the objects are not entirely contained within the borders. Much information for the latter case can be obtained from processing the object as usual and then applying statistical methods to determine possible matches with known objects. The other case is not as simple: Some type of image reduction is necessary if it is determined that an object is not known. Such software could selectively reduce protrusions of an object until a known object is identified.

5. ANALYSIS

To evaluate the use of the parallel architecture for computer vision, analytical comparisons of the parallel and serial algorithms were performed, and the simulation of the parallel software was compared to the serial implementation. An estimation of the computational speedups was derived by an examination of the structure of the parallel software. Table I summarizes the speedups for the major algorithms. The proportions of time required by different sections of the code were determined by executing a serial version of the algorithm. (The serial algorithm does not incur any overhead for operations such as transfers or processor disabling.) The time proportions are used to provide a weighting of the parallel speedup results. In this way, a section with low speedup that requires only a small fraction of the serial processing time does not falsely lower the overall speedup. Similarly, a section with high speedup that requires only a small fraction of the serial processing time does not falsely raise the overall speedup. With the time

Table 1
Computational performance results

Algorithm division	*Approx. speedup*	*Serial time*	*Time proportions*
Class()	$N(I/(I+N-1))$	15.36	0.3531
Holes()	$N/((N-1)(\text{SPIFAC}+1))$	15.79	0.3630
Areas()	(called by holes)	N/A	N/A
Center()	N	1.64	0.0377
Pstats()	N	10.71	0.2462

Note: I = Image border ($I \times I$ image); N = Number of PEs; SPIFAC = number of times a section of the object in the image can switch directions in crossing the image; for example, the letter Z would have a SPIFAC of 2. For the images analyzed, SPIFAC = 6.

Table 2
Parallel simulation: experimental results (64 × 64 image)[a]

Algorithm	*1 PE*	*2 PEs*		*4 PEs*		*8 PEs*	
	Avg	*Avg*	*Norm*	*Avg*	*Norm*	*Avg*	*Norm*
Class	40.25	42.25	21.13	50.25	12.63	58.67	7.333
Holes and areas	48.75	79.75	39.88	183.0	45.75	642.0	80.25
Center	5.25	6.25	3.125	6.5	1.625	7.0	0.875
Pstats	14.75	15.25	7.625	12.75	3.188	15.33	1.917
Time subtotal	109.0		71.76		63.19		90.38
Partial speedup	1		1.52		1.72		1.21
Chain (serial)	64.75	23.75	23.75	28	28	27.67	27.67
Chain (parallel)	N/A	76.75	38.38	139.25	34.81	272.33	34.04
Total time	173.75		133.89		126.0		152.09
Overall speedup	1		1.30		1.38		1.14
Efficiency	1		0.65		0.345		0.143

[a] *Times in 1 /60th second*

proportions, the total weighted speedup $S(N)$ for processing an $I \times I$ image using N PEs can be computed:

$$S(N) = \frac{0.3531NI}{I + N - 1} + \frac{0.3630N}{(N - 1)(SPIFAC + 1)} + 0.0377N + 0.2462N$$

$$= N\left[\frac{0.3531I}{I + N - 1} + \frac{0.3630}{(N - 1)(SPIFAC + 1)} + 0.2839\right]$$

The experimental results for the major sections of the software are presented in Table 2. The columns labeled Avg give the average time the serial simulation took for each step of the algorithm. The columns labeled Norm give the conversions of the average serial times to the average parallel times. This is the normalized execution time. The Time-subtotal row indicates how much time the first four component algorithms (internal/external classification, hole identification assuming a fixed number of passes, center of mass and perimeter statistics) required. The speedup that these partial times indicate is presented in the Partial-speedup row. The final algorithm step, formation of the chain code representation of the perimeter, is represented by two rows in the tables, because it has both a serial and a parallel component. Finally, the Total-time and Overall-speedup rows indicate the time that the entire processing operation needed and the speedup reflected by this time.

One additional measure of the performance of a parallel algorithm is the *efficiency* $E(N)$, defined to be the ratio of the speedup to the number of

processors [19]. Table 2 also shows the speedup for the case of a 64 × 64 image. For the example, although the speedup increases with N, the rate of increase is not proportional to N, and the efficiency decreases fairly sharply with N.

The simulations were designed to provide a conservative estimate of the speedup; assumptions about transfer timings and synchronization delays were approximated. The problem of nondeterminism in speedups was handled by using deterministic versions of nondeterministic routines. Again, these routines were designed to provide a conservative estimate of the speedup. No overlap of processing and transfers was assumed, although in many situations, inter-PE transfers can be performed at the same time that independent processing is occurring. The simulation results can, therefore, be used as a rough indicator of the speedup obtained by the parallel algorithms. Both the analytic and experimental results bear out the observation that the speedup will not grow as N, because the algorithms in which the largest proportion of time is spent (hole merging and chain code formation) have less than ideal speedup. (The experimental speedups are somewhat less than the analytic speedups due to the conservative assumptions made throughout the simulation.) In particular, the discrepancy between the theoretical and the experimental results is primarily in the holes and areas section. In this section, the theoretical results take into account the number of times the merging must be performed but do not take into account the overhead incurred by the transfers required by the merging. This overhead turns out to be a substantial portion of the algorithm, to the extent that it destroys the effectiveness of the increased parallelism. It appears that having subimages less than 16 pixels wide is counterproductive.

To address the problems with the hole merging algorithm, a new version of this algorithm was constructed that performs only the required number of hole merging steps (thus removing one of the earlier conservative assumptions). The algorithm is divided into two parts, which correspond to single-sided hole merging (such as was illustrated earlier) and multiple-edged hole merging (which handles ringlike holes such as the background hole). Each of these stages proceeds until the number of holes merged in each PE is zero. This has the advantage of eliminating unneeded overhead as well as having the capability of dealing with pathological cases that might require additional merging steps.

The results for this software with this modification included are in Table 3. Note that with this modification, for 64 × 64 images, eight processors still provide speedup gains, whereas previously only two or four could be used before the results deteriorated due to the overhead of the parallelism. With eight processors, the stripes in each PE are only eight pixels wide, so the proportion of time spent in overhead to coordinate between PEs is

Table 3
Non-deterministic merging parallel simulation results (64 × 64 image)[a]

	1 PE	*2 PEs*		*4 PEs*		*8 PEs*	
Algorithm	*Avg*	*Avg*	*Norm*	*Avg*	*Norm*	*Avg*	*Norm*
Class	40.25	45.25	22.63	49.0	12.25	58.67	7.333
Holes and areas	48.75	83.0	41.5	121.25	30.31	249.0	31.13
Center	5.25	6.5	3.25	6.75	1.688	6.67	0.8334
Pstats	14.75	15.75	7.875	14.0	3.5	15.33	1.917
Time subtotal	109.0		75.25		47.75		41.21
Partial speedup	1		1.45		2.28		2.64
Chain (serial)	64.75	24.0	24.0	28.0	28.0	28.33	28.33
Chain (parallel)	N/A	81.0	40.5	144.25	36.06	272.67	34.08
Total time	173.75		139.76		111.81		103.62
Overall speedup	1		1.24		1.55		1.68
Efficiency	1		0.62		0.388		0.210

[a] *Times in 1/60th second*

substantial. Thus, simulation demonstrated that the major problem with the parallel implementation is basically of one form: The number of transfers needed reduces the effectiveness of the parallelism. This can occur when the amount of information that is needed to make a proper decision (such as for hole merging) is large. This problem can manifest itself in several forms, such as algorithms that are inherently serial or that require data from the entire image. Such tasks might better be performed in one PE or in the control unit.

6. ARCHITECTURAL CONSIDERATIONS

A specific type of architecture has been assumed throughout this simulation and analysis. At this point, this restriction will be removed, and the tasks considered will be examined to explore a parallel architecture tailored to the characteristics of the vision task.

By examining the algorithms, one can see that a given memory area (the memory to be accessed in an interleaved manner, further improving system processing section. If the memory is dual-ported, with one *write* channel and two *read* channels, then the need for transfers can be virtually eliminated. In such an approach, the memory that was previously the exclusive responsibility of a specific PE would still be connected to that PE by the *write* channel and one of the *read* channels. However, the other *read* channel would be connected to a memory redirection network that would be setable by the

control unit when a new type of access pattern is needed. This redirection network could be either bidirectional or (more practical) two unidirectional networks, one direction being used to transmit the memory addresses and the other being used to return the data. The advantage of using two unidirectional networks is that information can be flowing in both directions at the same time without the need for redirection or buffering. This would allow the memory to be accessed in an interleaved manner, further improving system performance. When this scheme is compared with the number of transfers needed in some of the processing steps (such as in hole merging and Fourier descriptor preparation), the possible savings are evident.

7. SUMMARY

In this paper, analytic and simulation results for the application of parallel processing to the computer vision task have been presented. Because of the modular design of the software developed, it is possible to expand the processing sequence to include other common image processing techniques. From the analytic and simulation capabilities described, given specific speed requirements for a particular vision task and assumptions about processor speed, it will be possible to determine the number of processors needed to satisfy the task requirements. This work contributes to the understanding of the design of parallel systems for image processing applications.

REFERENCES

[1] Rice, T. A., and Siegel, L. J. (1983). Parallel algorithms for computer vision, *1983 Comp. Soc. Workshop on Computer Arch. for Pattern Analysis and Image Database Management*, 2, October, pp. 93–100.

[2] Flynn, M. J. (1966). Very high-speed computing systems, *Proc. IEEE*, 54, December, pp. 1901–1909.

[3] Pease, M. C. (1977). The indirect binary *n*-cube microprocessor array, *IEEE Trans. Comp.*, C-26, pp. 458–473.

[4] Siegel, H. J. *et al.*, PASM: A Partitionable SIMD/MIMD system for Image Processing and Pattern Recognition, *IEEE Trans. Comp.*, Vol C-30, December, pp. 934–947.

[5] Batcher, K. E. (1980). The design of a massively parallel processor, *IEEE Trans. Comp.*, C-29, September, pp. 836–844.

[6] Duff, M. J. B. (1982). Parallel algorithms and their influence on the specification of application problems, In *Multicomputers and Image Processing: Algorithms and Programs* (K. Preston and L. Uhr, eds). Academic Press, New York.

[7] Wulf, W., and Bell, C. (1972). C.mmp—A multi-miniprocessor, *AFIPS 1972 Fall Joint Comp. Conf.*, December, pp. 765–777.

[8] Swan, R. J., *et al.* (1977). The implementation of the Cm* multimicroprocessor, *AFIPS 1977 Nat'l. Comp. Conf.*, June, pp. 645–655.
[9] Gottlieb, A., *et al.*, (1983). The NYU ultracomputer—designing an MIMD shared memory parallel computer, *IEEE Trans. Comp.*, **C-32**, February, pp. 175–189.
[10] Siegel, H. J. (1984). *Interconnection Networks for Large Scale Parallel Processing: Theory and Case Studies*, D. C. Heath and Co., Lexington, Massachusetts.
[11] Kernighan, B. W., and Ritchie, D. M. (1978). *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey.
[12] Siegel, H. J. (1977). Analysis Techniques for SIMD Machine Interconnection Networks and the Effects of Processor Address Masks, *IEEE Trans. Comp.*, Vol. **C-26**, February, pp. 153–161.
[13] Goble, G. H., and Marsh, M. H. (1982). A Dual Processor Vax** 11/780, *IEEE 9th Annual Symp. on Comp. Arch.*, April, pp. 291–298.
[14] Nitzan, D., *et al.*, (1979). Machine intelligence research applied to industrial automation, SRI Report, Menlo Park, California, August.
[15] Wallace, T. P., and Mitchell, O. R. (1980). Analysis of three-dimensional movement using Fourier descriptors, *IEEE Trans. Pattern Analysis and Machine Intelligence*, **PAMI-2**, November, pp. 583–588.
[16] Arnold, K. Screen Updating and Cursor Movement Optimization, A Library Package, Unix* Version 4.1 bsd.
[17] Stone, H. S., ed. (1980). *Introduction to Computer Architecture*, Science Research Associates, Chicago, Illinois, pp. 394–396.
[18] Reynolds, D. E., and Otto, G. P. (1981). Software tools for CLIP4. Report No. 82/1, Dept. of Physics and Astronomy, University College, London, January.
[19] Kuck, D. J. (1977). A survey of parallel machine organization and programming, *Computing Survey*, **9**, March, pp. 29–59.

* Unix is a trademark of Bell Laboratories.
** VAX is a trademark of Digital Equipment Corporation.

Paper 3

A Parallel Algorithm for Contour Extraction: Advantages and Architectural Implications

From the Proceedings of IEEE Computer Society Conference on Computer Vision and Pattern Recognition, June 1983.

A PARALLEL ALGORITHM FOR CONTOUR EXTRACTION: ADVANTAGES AND ARCHITECTURAL IMPLICATIONS

David Lee Tuomenoksa†
George B. Adams III††
Howard Jay Siegel
O. Robert Mitchell

Purdue University
School of Electrical Engineering
West Lafayette, Indiana 47907

Abstract

Contour extraction is used as an image processing scenario to explore the advantages of parallelism and the architectural requirements for a parallel computer system, such as PASM. Parallel forms of edge-guided thresholding and contour tracing algorithms are developed and analyzed to highlight important aspects of the scenario. Edge-guided thresholding uses adaptive thresholding to allow contour extraction where gray level variations would not allow global thresholding to be effective. Parallel techniques are shown to eliminate some types of overhead associated with serial processing, offer the possibility of improved algorithm capability and accuracy, and decrease execution time. The implications that the parallel scenario has for machine architecture are considered. Various desirable system attributes are established.

I. Introduction

Image processing has long been an application viewed as suited to parallel processing [3]. Many individual image processing algorithms and their formulations for parallel processing environments have been studied, such as image coding [17], image correlation [1,25], image segmentation [6], two-dimensional FFT [18], histogramming [24], and line segment generation [26]. However, little work exists in considering a scenario as a whole for parallel processing. One such scenario is contour extraction. Contour extraction is a key tool for use in applications ranging from computer assisted cartography to industrial inspection.

In the past, edge information has been used to improve threshold selection [15] in the contour extraction process. A new scheme for determining threshold values has been developed by Suciu and Reeves [28]. This scheme has been incorporated in an image shape analysis method directed toward classifying small well-defined regions, such as buildings and airplanes, which has been investigated by Mitchell, Reeves, and Fu [16]. A processing scenario (composed of serial algorithms) which produces interpretation results from digitized imagery using these methods has been implemented at Purdue University on a VAX 11/780. In this application, image sizes are typically 5000-by-5000 *pixels (picture elements)*. The image is analyzed in 256-by-256 pixel subimages which are processed independently. To insure that each object (which has a maximum dimension of 127 pixels) will be completely contained within at least one subimage, it is necessary to overlap the subimages.

The serial method of [16] yields good results, but is computationally intensive, incurring long execution times. The time required to complete the processing scenario can be reduced by exploiting its inherent parallelism. In this work, a processing scenario composed of parallel algorithms which allows the problem to be completed with significantly reduced execution time is considered. In addition to decreasing the processing time, the parallel scenario does not place a limit on the maximum size of an object. Once it has been constructed, requirements the parallel scenario imposes on the architecture of a parallel computer system such as PASM [24] are studied.

A parallel computer system model is given in Section II. In Section III the object shape analysis problem [16] is defined and the parallel scenario is overviewed. In Sections IV and V the parallel algorithms which compose the scenario are presented, and they are evaluated in Section VI. The implications the scenario has concerning system architecture are considered in Section VII.

II. SIMD/MIMD Model

An SIMD/MIMD machine (e.g., CAIP [12]) consists of a control unit, an interconnection network, and N processing elements (PEs), where each PE is a processor/memory pair. This is shown in Fig. 1. An SIMD/MIMD machine can operate in either *SIMD (single instruction stream - multiple data stream)* [8] or *MIMD (multiple instruction stream - multiple data stream)* [8] modes and can dynamically switch between them. When operating in SIMD mode, the control unit broadcasts instructions to all processors and each active processor executes the instructions on data in its own memory. The same instruction is executed simultaneously in all active processors. The interconnection network provides interprocessor communication. When operating in MIMD mode, each processor fetches instructions from its own memory and executes them on data in its own memory. In MIMD mode, the control unit may coordinate the activities of the PEs. A partitionable SIMD/MIMD system (e.g., PASM [24], TRAC [11,20]) can be dynamically reconfigured to operate as one or more independent SIMD/MIMD machines of varying sizes. In this paper

This research was supported by the United States Army Research Office, Department of the Army, under grant numbers DAAG29-81-K-0088 and DAAG29-82-K-0101; the Defense Mapping Agency, monitored by the United States Air Force Command, Rome Air Development Center, under contract number F30602-C-0193; and by the Air Force Office of Scientific Research, Air Force Systems Command, USAF, under grant number AFOSR-78-3581. The United States Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation hereon.

† D. L. Tuomenoksa is now with American Bell, Holmdel, New Jersey 07733.

†† G. B. Adams III is now with the Research Institute for Advanced Computer Science, NASA Ames Research Center, Moffett Field, California 94035.

CH1891-1/83/0000/0336$1.00 © 1983 IEEE

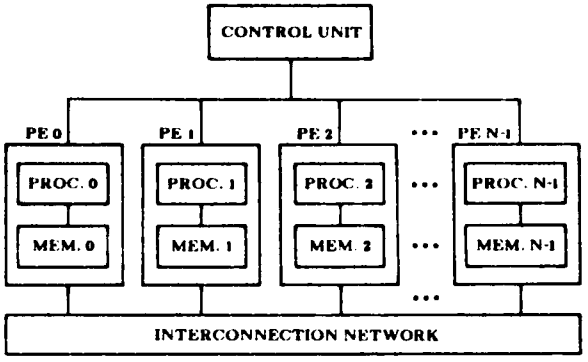


Fig. 1. Model of an SIMD/MIMD machine.

PASM is used as an example parallel computer system.

PASM, a partitionable SIMD/MIMD machine, is a large-scale dynamically reconfigurable multimicrocomputer system being designed at Purdue University [23,24]. Image processing and pattern recognition tasks are the target problem domain for PASM, and the requirements of these applications are being used to guide design decisions. PASM is intended to be a flexible research machine, and it has more capability than is necessary to cope with the example image processing scenario discussed in this paper. In particular, PASM's capability to be partitioned to operate as many independent SIMD/MIMD machines of varying sizes is not needed for this scenario.

The rest of this section is a brief overview of PASM to provide background for the following sections. A block diagram showing the basic components of PASM is given in Fig. 2. The *System Control Unit* is a conventional machine, such as a PDP-11, and is responsible for the overall coordination of the activities of the other components of PASM. The *Parallel Computation Unit (PCU)* contains $N = 2^n$ processors, N memory modules, and an interconnection network. The *PCU processors* are microprocessors that perform the SIMD and MIMD computations. The *PCU memory modules* are used by the PCU processors for data storage in SIMD mode and both data and instruction storage in MIMD mode. PASM is being designed for N = 1024. An N = 16 prototype based on Motorola MC68000 processors is planned [13].

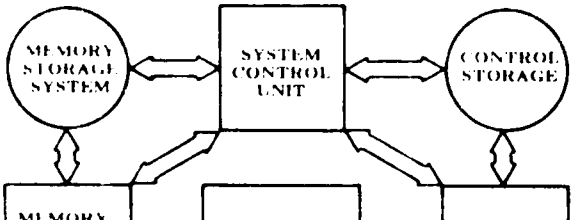


Fig. 2. Block diagram overview of PASM.

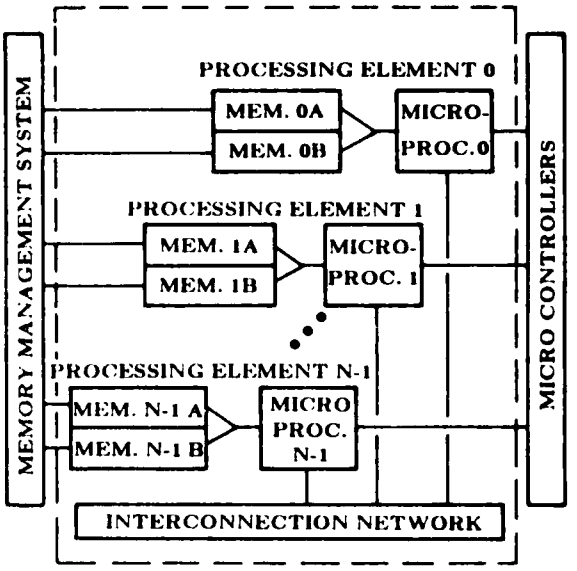


Fig. 3. PASM Parallel Computation Unit.

The PCU is organized as shown in Fig. 3. A pair of memory units is used for each PCU memory module so that data can be moved between one memory unit and secondary storage while the PCU processor operates on data in the other memory unit (double-buffering). Each memory unit is of substantial size (e.g., 64K words). A processor and its associated memory module form a *PCU PE*. The PCU PEs are addressed (numbered) from 0 to N−1. The *interconnection network* provides a means of communication among the PEs. PASM will use either an Extra Stage Cube type [2,22] or Augmented Data Manipulator type [14,21] of multistage network. The *Memory Management System* controls the loading and unloading of the PCU memory modules from the multiple secondary storage devices of the *Memory Storage System*.

The *Micro Controllers (MCs)* are a set of microprocessors which act as the control units for the PEs in SIMD mode and orchestrate the activities of the PEs in MIMD mode. *Control Storage* contains the programs for the MCs.

III. Image Processing Task

A. Problem Definition and Serial Algorithms

The first stage of the shape analysis scenario of [16] is to identify boundaries of potential objects using edge-guided thresholding [28]. Edge-guided thresholding (EGT) uses adaptive thresholding to allow contour extraction where gray level variations would not allow global thresholding to be effective. The image is segmented by selecting several gray level thresholds and tracing the resulting contours. Classification is accomplished by comparing the contours with prototype object models using either Fourier descriptors [30] or standard moments [10,29].

An overview of the serial image processing scenario follows (further details are given later in this section). Segmentation is simplest when there is little background information, i.e., the objects of interest cover a significant portion of the image. To achieve this with a very large

image, the image can be divided into subimages. A subimage size twice the largest dimension of an object is chosen, and each subimage is processed independently. Subimages are located so that they overlap neighboring subimages 50 percent in both the horizontal and vertical direction. This insures that an object will be completely contained in at least one block. However, it is necessary to perform the image processing computations four times for each pixel. The advantage of this method is that it eliminates the need to trace contours across subimage boundaries (simplifying the algorithms) and significantly reduces the amount of main memory required (subimages are discarded after processing).

Potential thresholds for a subimage are selected using edge-guided thresholding, which selects thresholds based on an edge-matching criterion. Using the Sobel edge operator [7], an *edge image* is generated in which gray levels indicate the magnitude of the gradient. A figure of merit which indicates how well a given thresholded gray level image matches edges in the edge image is then computed for every possible threshold. Using thresholds with high figures of merit, a requantized version of the gray level image is generated. A median filter [9] may then be applied to remove isolated noise artifacts. The contours for all potential objects not touching the subimage boundary (i.e., completely contained within the subimage) are extracted for further shape analysis. Very short and very long contours may not be retained if they represent objects outside the range of interest. The boundary of each object (contour) is stored as a sequence of x-y coordinates.

B. Parallel Scenario

In this section a parallel formulation of the contour extraction scenario is presented. This parallel scenario will be used as an application example for determining the execution environment which must be provided by the architecture of an SIMD/MIMD parallel processing system such as PASM. The specific context of the contour extraction scenario would depend on the application. The contour extraction scenario may be preceded by image processing such as rectification. Subsequent use of the extracted contours depends on the particular end application. Highlighting contours of an image requires essentially no further processing, while shape analysis and classification may involve significant additional calculation beyond contour extraction.

An M-by-M pixel image is represented by an array of M^2 pixels, where the value of each pixel is assumed to be an eight-bit unsigned integer representing one of 256 possible gray levels. To implement contour extraction on an SIMD/MIMD machine of 1024 PEs, assume that the PEs are logically configured as a 32-by-32 grid, on which the M-by-M image is superimposed, i.e., each processor has an M/32-by-M/32 subimage (see Fig. 4(a)). For M = 5120, each PE stores a 160-by-160 subimage. Each pixel is uniquely addressed by its i-x-y coordinates, where x and y are the x-y coordinates of the pixel in the subimage contained in PE i.

Two important parallel algorithms of the contour extraction scenario are edge-guided thresholding and contour tracing. The edge-guided thresholding algorithm, discussed in Section IV, is used to determine a set of optimal thresholds for each subimage. The contour tracing algorithm, which is considered in Section V, uses the set of optimal thresholds to segment the image and trace the contours, generating an i-x-y sequence for each contour.

The parallel algorithms described yield a significant

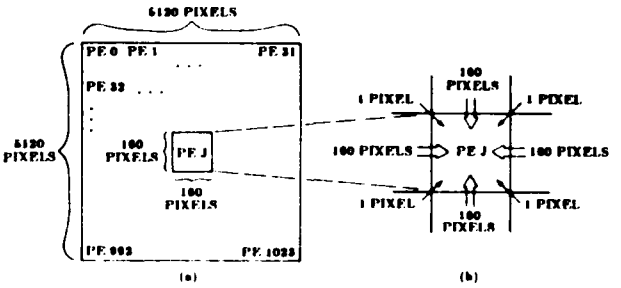


Fig. 4. (a) Data allocation for a 5120-by-5120 image using 1024 PEs.

(b) Data transfers needed to apply Sobel edge operator.

reduction in execution time because the multiplicity of processors allows all of the subimages to be processed simultaneously. Since the parallel contour tracing algorithm is able to trace contours over subimage borders, it is not necessary to overlap the subimages, and each pixel is processed only once. The parallel algorithms can result in improved information extraction since the subimages can be smaller (assuming a large number of PEs), yielding a better choice of thresholds within each subimage. In addition, the parallel algorithms do not require an object to be contained in a single subimage.

The parallel scenario could be implemented on a serial computer system with virtual memory [4]. The disadvantage of this approach is that when a contour spans more than one subimage, the linking of partial contours residing in different subimages requires that a representation of the subimages, as well as any contour information, be accessible. This may result in significant delay due to paging subimages into primary memory. Paging overhead does not occur on a parallel system since the entire image is stored in primary memory. Thus, it is the multiplicity of primary memories in a parallel system such as PASM (the large primary memory space) that makes the non-overlapping subimage approach practical.

IV. Edge-Guided Thresholding

The first major procedure of the example scenario is *edge-guided thresholding (EGT)* [28], which is used to identify boundaries of possible objects. Edge-guided thresholding selects threshold levels based on an edge-matching criterion instead of the classical technique of image histogram local minimum values [19]. Frequently, EGT gives better results than the histogram method because it is able to detect small regions not discernibly represented in the histogram [28].

The EGT algorithm operates on each subimage independently, and consists of three major steps. First an edge image is generated. Then a figure of merit is computed for every possible threshold. Finally, local maxima (peaks) in the figure of merit function determine the threshold levels.

The Sobel edge operator is used to generate the edge image in the example scenario. SIMD parallelism is the most advantageous form of parallelism for the Sobel algorithm. This can be shown by analysis of the operator itself. Let the image I be M-by-M and I(x,y) be a gray level image pixel, where $0 \leq x, y \leq M-1$. The Sobel procedure (ignoring image edge pixels for clarity) is the following.

```
for x = 1 to M-2 do
  for y = 1 to M-2 do
    sx(x,y) = 1/4 |(I(x-1,y-1) + 2*I(x-1,y) + I(x-1,y+1))
                 -(I(x+1,y-1) + 2*I(x+1,y) + I(x+1,y+1))|
    sy(x,y) = 1/4 |(I(x-1,y-1) + 2*I(x,y-1) + I(x+1,y-1))
                 -(I(x-1,y+1) + 2*I(x,y+1) + I(x+1,y+1))|
    g(x,y) = sqrt(sx(x,y)^2 + sy(x,y)^2)
```

The value g(x,y) represents the gradient at pixel (x,y), and these values form the edge image. The M-by-M image in the Sobel operator definition corresponds to a subimage within a PE for the scenario.

The algorithm is particularly well suited for SIMD parallelism because all pixels are processed identically. This complete synchronization aids the PE-to-PE communication necessary when subimage border pixels within each PE must be processed. In the case of this algorithm, transmission delays incurred due to PE-to-PE data transfers can be overlapped with data processing to reduce total execution time. All PEs will simultaneously request the same border pixel relative to their subimages. For example, when processing begins (with the upper left corner subimage pixel) all PEs will request (from the PE to their upper left) the pixel immediately above and to the left of their upper left corner pixel (if this pixel is within the complete image). This transfer of data from upper left neighbors can occur for all PEs simultaneously. A total of $4*(160 + 1) = 644$ parallel transfers are needed for a 5120-by-5120 pixel image, as shown in Fig. 4(b). The candidate interconnection networks for PASM can support these parallel transfers from any neighboring PE. The result of the Sobel operator is the edge image. High edge image pixel values indicate the presence of an edge.

The next step of the EGT algorithm is to compute a figure of merit value for each possible gray level. The figure of merit is a measure of how well the edges generated by a given threshold match the edges detected by the Sobel operator. Specifically, the figure of merit is determined as follows.

1. The local maximum and minimum pixel values over a 3-by-3 window are determined for each gray level image pixel.
2. For each possible threshold value (i.e., all gray levels) the center pixel of the 3-by-3 window is tested to see if it is an *edge point*. It is an edge point if the threshold is greater than or equal to the local minimum and less than the local maximum.
3. The mean of the edge image pixels corresponding to the gray level image pixels found to be edge points at a given threshold is the figure of merit for that threshold.

The figure of merit calculation has portions suited to both SIMD and MIMD parallelism. Steps 1 and 2 can be done efficiently in SIMD mode since all pixels are processed similarly. Step 3 is executed only on the gray level image pixels which are edge points. To do this, the PEs operate in MIMD mode, each sequencing through the edge points in its subimage. Since the number of such pixels may vary, some PEs may complete Step 3 before others.

The greater the mean of the edge points in Step 3, the better the match between threshold-generated boundaries and the edges detected by the Sobel operator. To avoid the assignment of a high figure of merit to a small number of noise pixels, a bias can be added to the denominator when calculating the mean. This has the effect of lowering the figure of merit if only a small number of pixels are above the threshold. The gray levels associated with local maxima (peaks) in the figure of merit function are chosen for image segmentation. Typically, three to six levels are chosen. The next step of the scenario is contour tracing.

V. Contour Tracing

In this section an approach to performing contour tracing using MIMD parallelism is presented. Initially, each PE contains a list of threshold values, $\{T_1, T_2, ..., T_t\}$, for its subimage which have been selected using edge-guided thresholding. The number of thresholds for any given PE is denoted by t and can differ for each PE. The contour tracing algorithm has two phases. In Phase I, the subimage is segmented within each PE and all local contours (both closed and partial) are traced and recorded. In Phase II, the partial contours traced during Phase I are connected.

A contour table is constructed in each PE containing an entry for every contour, whether partial or closed, which is located in the subimage associated with that PE. Each contour table entry contains the following fields: (a) a contour identification number, (b) the threshold value which generated the contour, (c) the number of pixels in the contour, (d) a flag indicating if the contour is closed or partial, (e) a pointer to the array containing the i-x-y sequence of the contour, (f) a flag indicating whether the partial contour has been connected (for use in Phase II), (g) the physical address of the PE which linked the contour, (h) the physical PE address and identification number denoting the partial contour blocking extension of the contour, and (i) a locked/unlocked semaphore. Contour table entries g, h, and i are discussed below. Each PE also contains a *partial contour list*. This list has an entry for each partial contour containing the i-x-y coordinates of its two end points and a pointer to its contour table entry.

In Phase I there is no PE-to-PE communication. Each PE considers its threshold values T_i, $1 \leq i \leq t$, independently. Its subimage is segmented using each threshold level T_i. To create the segmented image for threshold T_i, pixels in the original image which have a value greater than or equal to T_i are assigned a value of one, while those which are less than the threshold are assigned a value of zero.

Contour tracing begins by scanning rows of the segmented image beginning with the top row. Scanning stops when a pixel with value one is found which has a zero-valued neighbor to either side. This pixel is marked as the *start point* of a new contour, and its i-x-y coordinates are stored. Consider this pixel as the center pixel of the 3-by-3 window in Fig. 5. The contour is traced in a counterclockwise direction generating a sequence of i-x-y coordinates. Beginning with the neighboring pixel in position five (see Fig. 5) and incrementing by 1 modulo 8 to determine the next pixel, the algorithm looks for a pixel which has a value of one. The algorithm stores the direction, p, of this new pixel and appends its i-x-y coordinate to the contour sequence. Treat this new pixel as the center point of the 3-by-3 window in Fig. 5. The algorithm then looks for the next pixel in the contour beginning with the pixel in position $(p + 5)$ modulo 8 (to produce a counterclockwise trace). Tracing continues

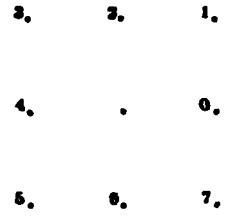


Fig. 5. Naming convention for the neighbors of the center pixel in a 3-by-3 window.

until the start point or a point of indecision is reached. If all of the neighbors of a start point are zero, that pixel is an isolated point and is ignored.

A *point of indecision* occurs when information from an adjacent subimage is required to determine the direction of the contour. When a point of indecision is reached, it is recorded as an *end point*, and the algorithm returns to the start point to trace the contour in a clockwise direction until another point of indecision is reached. When tracing in the clockwise direction, the new contour pixels are inserted onto the front of the i-x-y sequence. Each pixel in the contour is marked in the thresholded image so that the contour will not be retraced.

Consider the following contour tracing example based on Fig. 6. A 10-by-20 image is divided into two 10-by-10 subimages; each subimage is loaded into one of two PEs. The local threshold value T_1 is applied to the subimage in each PE. Each PE i begins scanning its respective subimage at pixel (i,0,0), for a one (indicated by a dot) with a zero on either side. PE 0 locates the edge of a segmented object at pixel (0,3,3). Pixel (0,3,3) is the start point for the new contour. PE 0 traces the contour of the object counterclockwise to a point of indecision at pixel (0,7,9), which is recorded as an end point. Pixel (0,7,9) is a point of indecision since pixels (1,6,0), (1,7,0), and (1,8,0) of the subimage in PE 1, which could extend the contour, are not in the subimage contained by PE 0. PE 0 then traces the contour in the clockwise direction beginning at pixel (0,3,3), reaching a point of

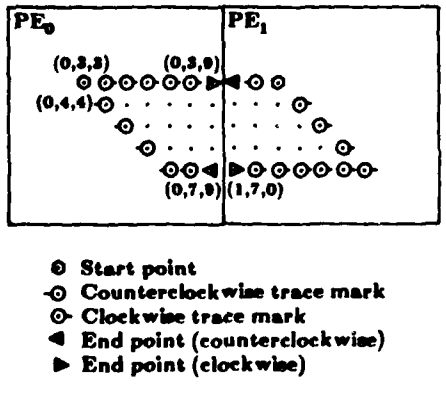


Fig. 6. Example of Phase I contour tracing for a 10-by-20 image. The triple (i,x,y) represents the i-x-y coordinates of a pixel.

indecision at pixel (0,3,9). After the clockwise trace, the first pixel in the i-x-y sequence describing the contour is (0,3,9). PE 0 resumes scanning at pixel (0,3,4) and finds no other contours in its subimage. Note that, for example, pixel (0,4,4) is not a start point for a new contour since it was marked during the trace of the first contour. Similarly, a partial contour is located in PE 1 with (1,7,0) as the first pixel in its i-x-y sequence. Once a PE has scanned the segmented image generated by threshold T_i, it repeats the process for threshold T_{i+1}. After all threshold values in a PE have been considered, Phase I is complete.

In Phase II, each PE attempts to connect its partial contours to partial contours which are located in neighboring PEs. There are two alternatives for determining when a PE can enter Phase II. With the first, PEs are allowed to start Phase II processing after all have completed Phase I. With the second, a PE enters Phase II immediately after completing Phase I. However, it can only attempt to extend contours into subimages of PEs which are also in Phase II. If all neighboring PEs are still in Phase I, the PE must wait. The latter approach may reduce the total scenario execution time since the PE with the longest Phase I time may well not be the one with the longest Phase II time. The first alternative requires time equal to the sum of the longest times in each phase.

Since multiple PEs can contain portions of the same contour, there must be a rule to determine which PEs have priority to attempt to close a contour. The rule is each PE attempts to extend only its partial contours which have both end points bordering subimages to the left and/or above. For example, in Fig. 7, partial contours A, B, C, and D are considered by the PE, while E, F, and G are not. For each given partial contour (generated by a threshold T_j), the PE attempts to extend it into the neighboring PE from the counterclockwise end point (as described below).

In order for a PE to extend a contour, it must be able to access and modify contour tables which are located in other PEs. As a result, a mechanism to prevent one PE from using a contour table entry while another PE is in the process of using that entry must be provided by the system and used by the contour tracing algorithm. Any section of code which modifies a contour

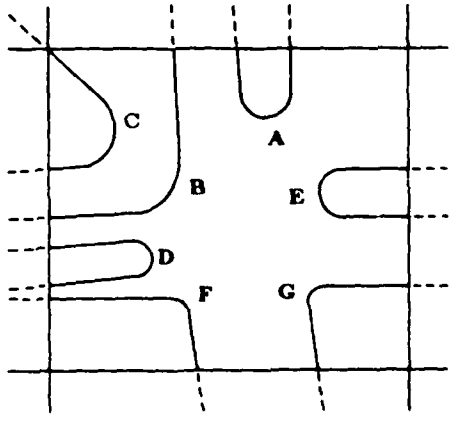


Fig. 7. Phase II connection precedence. Partial contours A, B, C, and D are considered by the PE; E, F, and G are not.

table entry is a *critical section* [5]. The only table entry fields which can be modified by another PE are the flag which indicates if the partial contour has been connected and the physical address of the PE which linked the contour (fields (f) and (g)). While a critical section is being executed on a given table entry, that entry is locked, so no other processor can modify it.

A *semaphore* is a variable whose value indicates whether or not a critical section can be entered [5]. There is a semaphore for each contour table entry which can take on a value of zero or one. Before a PE enters a critical section (for a given contour), the processor performs a *P-operation* [5] on the given contour to determine if it is unlocked. If the semaphore for the contour table entry is one, the processor sets the semaphore to zero (locking the contour table entry so that no other processor can access it) and enters the critical section, free to modify the contour table entry. When the processor completes modification of the contour table entry (i.e., the critical section ends), it performs a *V-operation* [5] on the semaphore for the contour, setting the semaphore to one. The contour table entry is then unlocked. On the other hand, if the semaphore is initially zero, the processor receives a message indicating that the partial contour is locked.

If the end point of a given partial contour is not at a corner of its subimage, there are three pixels, located in the adjacent subimage, which can possibly extend the contour. The PE accesses the partial contour list for the adjacent subimage (see Section VII). Considering the possible extending pixels one at a time in counterclockwise order, the PE checks the partial contour list to determine if any partial contours in the adjacent subimage have the possible extending pixel as an end point. If such a partial contour exists, the PE performs a P-operation on the contour table entry pointed to by the partial contour list. If the contour was unlocked, the i-x-y sequence for the contour is transferred (discussed in Section VII) to the PE containing the given partial contour and then concatenated to its i-x-y sequence, forming a new, extended partial contour. If there is more than one partial contour with the same end point which can extend the given contour, the partial contour which was generated by a threshold value closest to that for the given contour is selected.

If the end point of a given partial contour is a corner point of its subimage, there are five pixels located in adjacent subimages which can possibly extend the contour. Since these five pixels are located in three different subimages, the PE attempting to extend the given partial contour must check for continuation in each of the upper-left adjacent subimages (in a counterclockwise order).

Note that regardless of where partial contour end points lie, the search for pixels to extend the contour can be widened beyond the three or five pixels here to allow for threshold value discontinuities at subimage boundaries. Thresholds could be interpolated across subimage boundaries to allow partial contours with non-adjacent end points to be joined.

Assume that PE i has a partial contour which it is responsible for extending. If a continuation of the partial contour is not found in the partial contour list for the adjacent subimage, PE i probes into the adjacent subimage to determine if an extension of the partial contour can be generated by the threshold, T, it (PE i) used to trace its partial contour. If so, PE i extends its partial contour by accessing the data from the adjacent PE. Instead of creating an entire segmented subimage for the threshold T, PE i dynamically thresholds pixels as needed. This contour generation using T is done since it is possible that the partial contour in the adjacent PE was not located in Phase I because different threshold values were used, or the contour fell along the edge of the subimage (see the split between PEs 2 and 3 in Fig. 9).

Once PE i locates a partial contour in an adjacent subimage which continues the given contour and has stored the concatenated contour in its contour table, it repeats the process, if necessary, by following the contour to the next PE until the contour is closed or cannot be extended. A limit is placed on the maximum contour length to guarantee algorithm termination in the event of a pathological image.

Consider the example in Fig. 8 where a 12-by-12 pixel image is divided between four PEs. After Phase I,

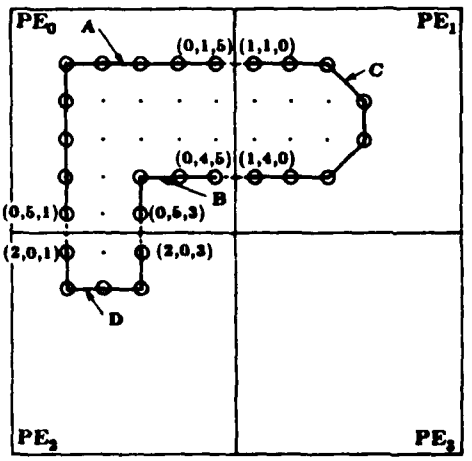


Fig. 8. Example where two PEs attempt to close the same contour. End point coordinates are given where (i,x,y) represents the i-x-y coordinates of the pixel.

PE 0 contains partial contours A with end points (0,1,5) and (0,5,1) and B with end points (0,4,5) and (0,5,3); PE 1 contains partial contour C with end points (1,4,0) and (1,1,0); and PE 2 contains partial contour D with end points (2,0,1) and (2,0,3). Since both end points for contour C border the subimage to the left, PE 1 attempts to extend contour C in Phase II. Similarly, PE 2 attempts to extend contour D since its end points border the subimage above.

PE 1 attempts to extend C in the counterclockwise direction, i.e., from pixel (1,1,0). It first locks its contour table entry for C. It then examines the contour table of PE 0 and determines that A can be linked to C. If the table entry for A is unlocked (i.e., the semaphore value is one), PE 1 locks it (performs a P-operation) and appends the i-x-y sequence of A to the i-x-y sequence of C. It also sets the flag which indicates that A has been linked and records that PE 1 performed the linkage.

Independently of the actions of PE 1, PE 2 attempts to extend contour D (from pixel (2,0,3)). As did PE 1 with A, PE 2 appends B to D. If PE 2 attempts to extend the result, DB, while PE 1 is in the process of extending C into PE 0, it will find C locked. PE 2 then abandons its attempt to close the contour, since PE 1 is

also attempting to do it, and unlocks partial contour DB. This allows PE 1 to access DB after it has appended A to C. Therefore, the closed contour CADB is ultimately traced completely and stored by PE 1. If PE 1 had completed linking A to C before PE 2 completed linking B to D, the closed contour would have been completely traced by PE 2. *Deadlock* is the situation when each of two or more PEs are halted while waiting for the other(s) to continue [27]. If a PE is blocked due to a lock then (1) not allowing a PE to wait for access to a locked contour table entry of another PE, and (2) requiring the blocked PE to unlock its affected partial contour prevents deadlock.

If PE 1 and PE 2 had completed their first linking operation *simultaneously*, both would have abandoned tracing the contour (i.e., no PE would link the contour CADB). To insure that the linking of a contour will not be abandoned by all PEs, the following protocol is used. Assume PE i is blocked from extending a contour X by PE j, which has higher positional precedence (i.e., $i < j$). In that case, PE i unlocks contour X and sends a message informing PE j that PE i has abandoned its attempt to further extend contour X. If PE j had also abandoned the contour, this message would cause PE j to try again. The message sent from PE i to PE j contains the identification number of contour X and the value i. After receiving the message, PE j searches its contour table to determine if it abandoned X. To do this it uses field (h) of the contour table. For the above example PE 2 would link the partial contours since it has higher precedence.

Deadlock with multiple contours cannot occur since each PE considers only one contour at a time and does not abandon the attempt to extend that contour until that PE has closed the contour or has relinquished control to another PE to close that contour.

When Phase II of the algorithm is complete, the i-x-y sequence for each contour in the image will be contained in exactly one of the PEs which contained part of the contour originally. In the example given in Fig. 6, PE 1 will contain i-x-y sequence for the contour.

As a final example, a 30-by-20 image is divided into six 10-by-10 subimages, each subimage is loaded into one of six PEs. In Figs. 9 and 10 the results of Phase I and II processing are shown, respectively. Even though the entire object in PE 5 was located within the subimage, the left edge of the object was not traced in Phase I since PE 5 could not determine whether the object continued into the next subimage. On the other hand, a closed contour was found in Phase I for the object in PE 4 since the object did not include any border pixels of the subimage.

VI. Algorithm Evaluation

Subimage size for the serial algorithm is chosen to be twice the maximum allowed object dimension so that overlapping of subimages guarantees that each object appears in its entirety in some subimage. With this property, partial contours never need to be considered; all objects are found as closed contours within a subimage.

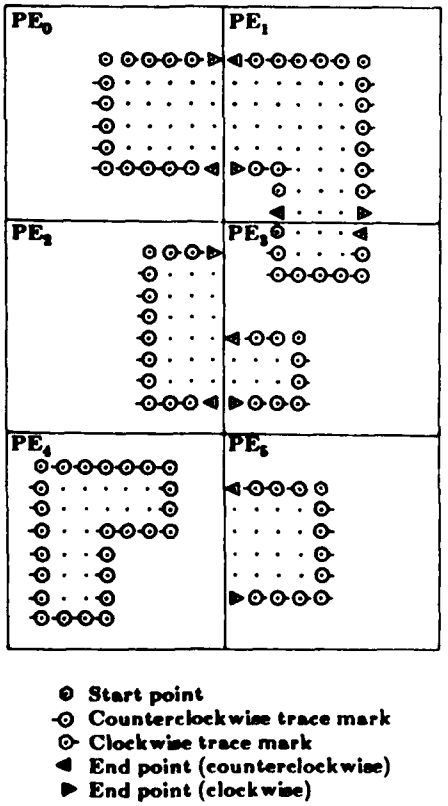


Fig. 9. Results of Phase I of contour tracing for a 30-by-20 subimage.

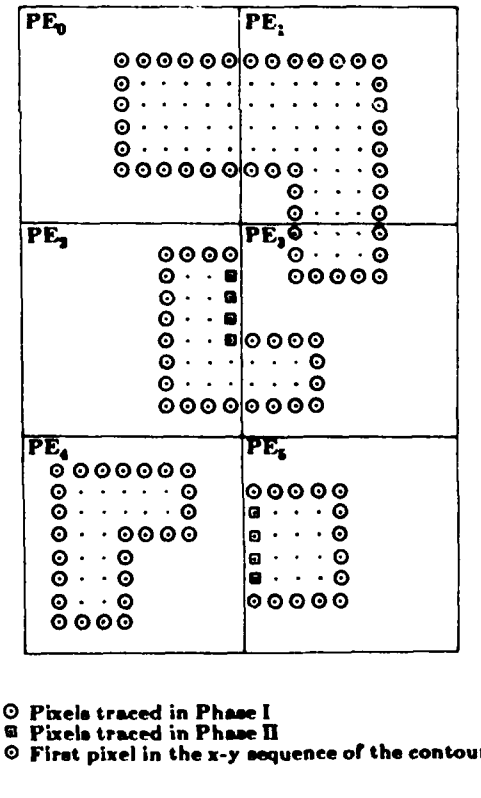


Fig. 10. Results of Phase II of contour tracing for a 30-by-20 subimage.

The advantage of the serial approach (Section III.A) over the parallel approach (Section III.B) is that partial contour extension is not necessary. The disadvantages of the serial approach when compared to the parallel approach are threefold. First, the maximum size of an object of interest must be established so that subimage size is known. This choice is constrained by the fact that EGT performance tends to degrade with increasing subimage size. Thus, there is a practical limit on the maximum object size. Second, each pixel is processed for contour extraction four times. Finally, thresholding (including EGT) tends to perform less well when objects are small relative to the image (in this case, subimage) size. The parallel algorithms do not limit maximum object size, process each pixel just once, and may improve threshold accuracy by allowing ready use of small subimages. Thus, parallel systems can allow the full benefits of adaptive thresholding via EGT to be more readily realized.

Speedup is the usual rationale for employing parallel processing techniques, and the example parallel scenario has the potential for significant speedup. However, the speedup is data dependent. This is because the PE workload may be highly varied during contour tracing due to uneven distribution of contours throughout the image being processed. While it may be possible to implement load sharing for this portion of the scenario (with certain overhead costs), inequities reducing actual speedup are almost certain to remain.

Overall, the parallel algorithms presented are strong contenders to replace serial methods in some applications. One such is quality control inspection of printed circuit boards. In this application, large object handling capability is needed for following long circuit traces, and sufficient speedup is necessary for timely response. Other applications involve military environments where real-time processing is crucial.

VII. Architectural Implications

The study of a parallel formulation of an image processing scenario involves both the design of individual parallel algorithms and the determination of a method to integrate them into a single job. This leads to an understanding of necessary and useful hardware attributes for a parallel machine intended to execute that scenario. For the example scenario, aspects of each algorithm which have an architectural impact other than those pertaining to the processors will be listed. Processor specific considerations (e.g., instruction set) are not treated because they are similar for serial and parallel machines.

The Sobel edge detection algorithm step of EGT requires data that is, by vast majority, local to each PE. When non-local data is required, nearest neighbor PEs comprise the set of data sources. Local maxima and minima calculation on 3-by-3 windows mimics the characteristics of the Sobel operator, but with more memory references. Edge point detection is similar in these regards to the previous steps.

The figure of merit calculation for EGT is different in kind from the previous steps. Only local data is required, and processing time is data dependent. MIMD operation is preferable to SIMD, even if edge point detection and figure of merit calculations are merged into a one-pass operation.

Phase I of contour tracing requires only local data, but execution time is data dependent. Phase II makes heavy use of non-local data and has data dependent execution time. Both phases are suited to MIMD mode.

Now the architectural requirements for a parallel machine performing the example scenario can be considered. Probably the most basic need for the system if it is to support the scenario well, is to be capable of dynamically switching between SIMD and MIMD operation, as can PASM. With only SIMD capability, vast inefficiency would occur in later stages of the scenario. Having only MIMD mode is a less serious handicap, but will lengthen execution time for the Sobel operator and determining local maxima and minima, due to the need for explicit synchronism and data sharing. Thus, the capability to dynamically switch between SIMD and MIMD modes is important so that each subsequent portion of the scenario can be executed in the most appropriate operational mode.

An interconnection network is needed to perform permutations involving eight nearest neighbors in SIMD mode. In MIMD mode, it is used for eight nearest neighbors and for somewhat arbitrary one-to-one connections (when transferring partial contour information between non-adjacent PEs). Both types of connection needs must be performed efficiently by the network. The networks proposed for PASM can do so.

The PE-to-PE transfer of information must be efficient, or the parallel algorithms will be slowed. One method to perform PE-to-PE communication is by using direct memory access (DMA). DMA is a method for storing or retrieving data without processor intervention. There are several ways to implement this capability. In one, a PE extending a partial contour sends an interrupt to the remote PE containing the extension of the partial contour along with the identifier of the needed partial contour. The remote PE then enters a DMA handling routine. This routine computes the local memory address range of the requested partial contour i-x-y sequence and sends this information along with the requesting PE number to special DMA hardware. The DMA hardware then autonomously retrieves the information from local memory and performs necessary network interfacing to send the data to the requesting PE. DMA hardware accesses to local memory can be via cycle stealing. Another implementation of DMA capability is through an intelligent network interface unit (NIU). Requests for data from remote PEs would be received, interpreted, and discharged by the NIU without local PE processor intervention. The NIU would combine DMA capability with network protocol support. VLSI technology may allow ready fabrication of sophisticated NIUs. Such a capability would be worthwhile to include in a system such as PASM.

VIII. Summary

Considering an entire scenario in the light of parallelism is a useful approach for matching image processing tasks and parallel architectures. A number of observations were made and conclusions drawn from the example image processing scenario. In particular, the parallel scenario was found to embrace both SIMD and MIMD subtasks, involve significant PE-to-PE data transfer, and contain both nearest-neighbor and non-adjacent PE communication patterns. Parallel formulation of the algorithms lead to several advantages including speedup, elimination of object size constraints, and potential for improved accuracy. These observations indicate that parallel contour extraction could be useful in industrial inspection and military applications. They suggest desirable system architecture features, including SIMD/MIMD capability with dynamic mode switching, dedicated PE-to-PE communication support hardware, and arbitrary PE-to-PE interconnection capability. These requirements are consistent with the capabilities of PASM.

Acknowledgment

The authors gratefully acknowledge the comments made by Timothy A. Grogan and Dr. Robert J. McMillen.

References

[1] D. L. Ackerman, "Algorithm design for digital image correlation on a parallel processing system," in *High Speed Computer and Algorithm Organization*, edited by D. J. Kuck, D. H. Lawrie, and A. H. Sameh, Academic Press, Inc., New York, 1977, pp. 307-308.

[2] G. B. Adams III and H. J. Siegel, "The extra stage cube: a fault-tolerant interconnection network for supersystems," *IEEE Trans. Comput.*, vol. C-31, pp. 443-454, May 1982.

[3] J. A. Cornell, "Parallel processing of ballistic missile defense radar data with PEPE," *COMPCON '72*, Sept. 1972, pp. 69-72.

[4] P. J. Denning, "Virtual memory," *Computing Surveys*, vol. 2, pp. 153-188, Sept. 1970.

[5] E. W. Dijkstra, "Cooperating sequential processes," in *Programming Languages*, edited by F. Genuys, Academic Press, Inc., New York, 1968, pp. 43-112.

[6] R. J. Douglass, "A pipeline architecture for image segmentation," *15th Hawaii Int'l. Conf. on System Sciences*, Jan. 1982, pp. 360-367.

[7] R. O. Duda and P. E. Hart, *Pattern Classification and Scene Analysis*, Wiley, New York, 1973.

[8] M. J. Flynn, "Very high-speed computer systems," *Proc. IEEE*, vol. 54, pp. 1901-1909, Dec. 1966.

[9] N. C. Gallagher, Jr. and G. L. Wise, "Passband and stepband properties of median filters," *IEEE Trans. Acoustic Spch. Sig. Proc.*, vol. ASSP-29, pp. 1136-1141, Dec. 1981.

[10] M. K. Hu, "Visual pattern recognition by moment invariants," *IRE Trans. Info. Theory*, vol. IT-8, pp. 179-187, Feb. 1962.

[11] R. N. Kapur, U. V. Premkumar, and G. J. Lipovski, "Organization of the TRAC processor-memory subsystem," *AFIPS 1980 Nat. Comput. Conf.*, May 1980, pp. 623-629.

[12] J. Keng and K. S. Fu, "A special purpose architecture for image processing," *1978 IEEE Comput. Soc. Conf. Pattern Recognition Image Processing*, June 1978, pp. 287-290.

[13] J. T. Kuehn, H. J. Siegel, and P. D. Hallenbeck, "Design and simulation of an MC68000-based multimicroprocessor system," *1982 Int'l. Conf. Parallel Processing*, Aug. 1982, pp. 353-362.

[14] R. J. McMillen and H. J. Siegel, "Routing schemes for the augmented data manipulator network in an MIMD system," *IEEE Trans. Comput.*, vol. C-31, pp. 1202-1214, Dec. 1982.

[15] D. L. Milgram, "Region extraction using convergent evidence," *Computer Graphics and Image Processing*, vol. 11, pp. 1-12, Sept. 1979.

[16] O. R. Mitchell, A. P. Reeves, and K. S. Fu, "Shape and texture measurements for automated cartography," *1981 IEEE Comput. Soc. Conf. Pattern Recognition Image Processing*, Aug. 1981, pp. 367.

[17] T. N. Mudge, E. J. Delp, L. J. Siegel, and H. J. Siegel, "Image coding using the multimicroprocessor system PASM," *1982 IEEE Comput. Soc. Conf. Pattern Recognition Image Processing*, June 1982, pp. 200-205.

[18] P. T. Mueller, Jr., L. J. Siegel, and H. J. Siegel, "Parallel algorithms for the two-dimensional FFT," *5th Int'l. Conf. Pattern Recognition*, Dec. 1980, pp. 497-502.

[19] J. M. S. Prewitt and M. L. Mendelsohn, "The analysis of cell images," *Annals N.Y. Academy of Science*, vol. 128, pp. 1035-1053, 1966.

[20] M. C. Sejnowski, E. T. Upchurch, R. N. Kapur, D. P. S. Charlu, and G. J. Lipovski, "An overview of the Texas Reconfigurable Array Computer," *AFIPS 1980 Nat. Comput. Conf.*, May 1980, pp. 631-641.

[21] H. J. Siegel and R. J. McMillen, "Using the augmented data manipulator network in PASM," *Computer*, vol. 14, pp. 25-33, Feb. 1981.

[22] H. J. Siegel and R. J. McMillen, "The multistage cube: a versatile interconnection network," *Computer*, vol. 14, pp. 65-76, Dec. 1981.

[23] H. J. Siegel, P. T. Mueller, Jr., and H. E. Smalley, Jr., "Control of a partitionable multimicroprocessor system," *1978 Int'l. Conf. Parallel Processing*, Aug. 1978, pp. 9-17.

[24] H. J. Siegel, L. J. Siegel, F. C. Kemmerer, P. T. Mueller, Jr., H. E. Smalley, Jr., and S. D. Smith, "PASM: a partitionable SIMD/MIMD system for image processing and pattern recognition," *IEEE Trans. Comput.*, vol. C-30, pp. 934-947, Dec. 1981.

[25] L. J. Siegel, H. J. Siegel, and A. E. Feather, "Parallel processing approaches to image correlation," *IEEE Trans. Comput.*, vol. C-31, pp. 208-218, Mar. 1982.

[26] C. D. Stamopoulous, "Parallel algorithms for joining two points by a straight line segment," *IEEE Trans. Comput.*, vol. C-23, pp. 642-646, June 1974.

[27] H. S. Stone, "Parallel computers," in *Introduction to Computer Architecture*, 2nd edition, edited by H. S. Stone, Science Research Associates, Inc., Chicago, IL, 1980, pp. 363-425.

[28] R. E. Suciu and A. P. Reeves, "A comparison of differential and moment based edge detectors," *1982 IEEE Comput. Soc. Conf. Pattern Recognition Image Processing*, June 1982, pp. 97-102.

[29] M. R. Teague, "Image analysis via the general theory of moments," *Journal Optical Soc. Am.*, vol. 70, pp. 920-933, Aug. 1980.

[30] T. P. Wallace and P. A. Wintz, "An efficient three-dimensional aircraft recognition algorithm using normalized Fourier descriptors," *Comput. Graphics and Image Processing*, vol. 13, pp. 99-126, June 1980.

Paper 4

The Use and Design of PASM

From Integrated Technology for Parallel Image Processing, S. Levialdi, editor 1985, Academic Press, pages 133-152

Chapter Nine

*The Use and Design of PASM**

James T. Kuehn, Howard Jay Siegel, David Lee Tuomenoksa, and George B. Adams III

1. INTRODUCTION

Parallel processing has been successfully used to reduce the time of computation for a wide variety of applications. The processing of large amounts of data, the need for real-time computation, the use of computationally expensive operations, and other demands that would make a task too time-consuming to perform on conventional computer systems have forced computer architects to consider parallel/distributed computer designs. Applications that have one or more of these characteristic demands include image analysis for automated photo reconnaissance, map generation, robot (machine) vision, and rocket and missile tracking; digital signal processing for speech understanding and biomedical signal analysis; and vector processing for the solving of large systems of equations. To date, a variety of special-purpose machines has been constructed to speed the processing of select groups of algorithms. Examples are special-purpose digital signal processors such as the APS-II [1], array processors such as the AP-120B (Floating Point Systems, Inc. Portland, Oregon), and supercomputers with vector/pipeline operations such as the Cyber 205 [2].

Our goal is the design of a flexible parallel processing system that can be dynamically reconfigured to meet the particular processing needs of a large variety of applications in the image and speech analysis domains. The system

* The research was supported by the United States Army Research Office, Department of the Army, under grant number DAAG29-82-K-0101; by the United States Air Force Command, Rome Air Development Centre, under contract number F30602-83-K-0119; and by the National Science Foundation under grant ECS-81-20896.

being designed is a PASM, *p*artitionable *S*IMD/*M*IMD machine. In this chapter, two algorithms used in parallel contour extraction are given as an image processing scenario to explore the advantages and implications of using the PASM parallel processing system and to motivate the inclusion of its important architectural features. These features will help to identify the attributes of a custom-designed VLSI processor chip set for PASM. In particular, the architectural features that could be incorporated into a VLSI chip set that will match the needs of parallel algorithms in the image and speech processing domains will be explored. Using algorithm characteristics to drive the design of PASM will lead to a machine that has the necessary flexibility for executing image and speech processing algorithms.

In the next section, the parallel processing model and an overview of the PASM architecture are given. Section 3 outlines two algorithms of the contour-extraction task. The first algorithm, edge-guided thresholding, is discussed in Section 4. Section 5 describes the second algorithm, contour tracing. The architectural implications of these algorithms are explored in Section 6.

2. SIMD/MIMD MODEL

Two types of parallel processing systems are single-instruction stream–multiple-data stream (SIMD) machines and multiple-instruction stream–multiple-data stream (MIMD) machines [3]. A *SIMD machine* typically consists of a control unit, an interconnection network, and N *processing elements* (PEs), with each PE being a processor/memory pair (Fig. 1). The

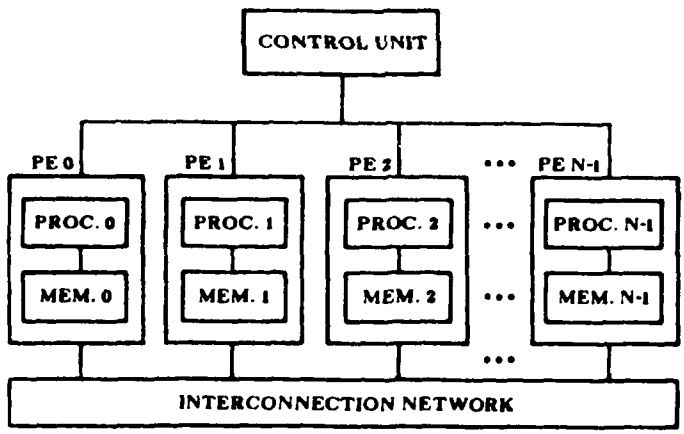


Fig. 1 Model of an SIMD/MIMD machine.

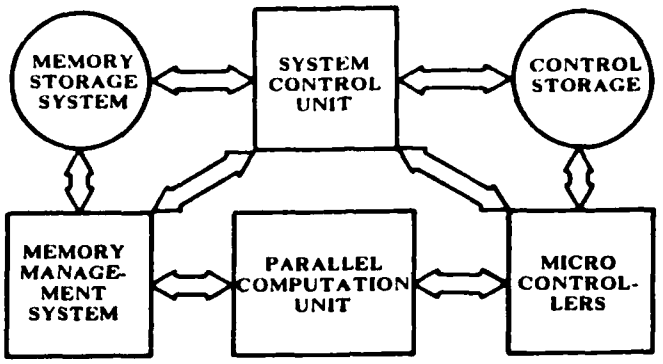


Fig. 2 Block diagram overview of PASM.

control unit broadcasts instructions to the processors, and all active (enabled) processors execute the same instruction at the same time. Each processor executes the instructions with data taken from its own memory. The interconnection network allows interprocessor communication. A *MIMD machine* has a similar organization, but each processor can follow an independent instruction stream. As with SIMD architectures, there is a multiple data stream and an interconnection network. The control unit may coordinate the activities of the PEs in MIMD mode. A SIMD/MIMD machine can operate in either mode and dynamically switch between them. A partitionable SIMD/MIMD system (e.g., PASM [4]; TRAC [5],[6]) can be dynamically reconfigured to operate as one or more independent SIMD/MIMD machines of various sizes.

PASM is being designed using a variety of applications problems from the areas of image and speech analysis to guide the machine design choices. It is not meant to be a production-line machine but a research tool for studying large-scale SIMD and MIMD parallelism.

A block diagram of the basic components of PASM is given in Fig. 2. The heart of the system is the *parallel computation unit* (PCU), which contains $N = 2^n$ processors, N memory modules, and an interconnection network. The PCU *processors* are microprocessors that perform the SIMD and MIMD computations. The PCU *memory modules* are used by the PCU processors for data storage in SIMD mode and both data and instruction storage in MIMD mode. The *interconnection network* provides communication among the PEs. PASM will use either an Extra Stage Cube type or Augmented Data Manipulator type of multistage network [7].

The PCU is organized as shown in Fig. 3. Each processor is connected to a memory module to form a PE. A pair of memory units is used for each

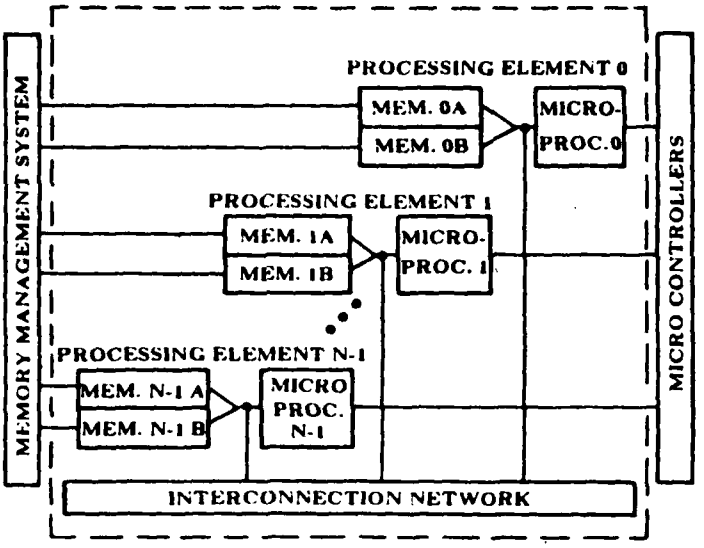


Fig. 3 PASM Parallel Computation Unit.

memory module. This double-buffering scheme allows data to be moved between one memory unit and secondary storage while the processor operates on data in the other memory unit. Each memory unit is of substantial size (e.g., 64K words). PEs are addressed (numbered) from 0 to $N - 1$.

The *system control unit*, a conventional computer, is responsible for the overall coordination of the activities of the other components of PASM. The *memory management system* controls the loading and unloading of the PE memory modules from the multiple secondary storage devices of the *memory storage system*. The *microcontrollers* (MCs) are a set of microprocessors that act as the control units for the PEs in SIMD mode and orchestrate the activities of the PEs in MIMD mode. Each of the Q MCs controls a fixed group of N/Q PCU PEs. By combining the effects of multiple MCs, virtual machines (partitions) can be created. *Control storage* contains the programs for the MCs. PASM is being designed for $N = 1024$ and $Q = 32$. An $N = 16$, $Q = 4$ prototype based on Motorola MC68000 processors is under development [8].

This brief overview of PASM provides the needed background for this chapter. Further details and a list of papers about PASM can be found in Siegel [9].

3. EXAMPLE TASK

Many individual image and speech processing algorithms and their formulations for parallel processing environments have been studied, such as 2-D FFTs [10],[11], Hadamard transforms [12], image correlation [13], histogramming [4], resampling [14], one-dimensional FFTs [15], linear predictive coding [16], and dynamic time warping [17]. However, rarely is a complete scenario considered as a whole. Consider the situation in which the results of one algorithm are used as input to another. In the parallel environment, this may strongly influence how each algorithm is structured. For example, results calculated in one PE might need to be communicated to another PE for use in a later algorithm.

Contour extraction is a key tool for use in applications ranging from computer assisted cartography to industrial inspection. Two algorithms from a contour extraction task will be used as an application example for demonstrating the architectural features that must be provided by PASM to have an appropriate execution environment. It will be shown how computational attributes of a parallel implementation of this example SIMD/MIMD scenario influence the hardware design choices, including those features that would be desirable in a custom-designed VLSI chip set.

The two algorithms to be considered are *edge-guided thresholding* (EGT) and *contour tracing*. The EGT algorithm, discussed in Section 4, is used to determine the optimal threshold for quantizing the image [18]. The contour-tracing algorithm, considered in Section 5, uses the set of optimal thresholds to segment the image and trace the contours. These two parallel algorithms are based on those developed in Tuomenoksa *et al.* [19] and are summarized here because their processing demands are quite different from each other. As will be seen, the EGT algorithm is best suited for SIMD mode, whereas MIMD mode will be used for the contour tracing algorithm. Also, the EGT algorithm will have inter-PE communication needs that are different from the communication needs of the contour tracing algorithm. Other aspects will be discussed in Section 6. For this task scenario, the ability to partition PASM is not used; i.e., all N PEs are employed.

4. EDGE-GUIDED THRESHOLDING

Consider an $M \times M$ pixel *input image* to be processed by the two algorithms. The value of each pixel is assumed to be an 8-bit unsigned integer representing one of 256 possible gray levels. Using the PASM model, assume that the PEs are logically configured as a $\sqrt{N} \times \sqrt{N}$ grid, on which the $M \times M$ image is superimposed; i.e., each processor has an

$M/\sqrt{N} \times M/\sqrt{N}$ *subimage*. For $M = 4096$, $N = 1024$, each PE stores a 128×128 subimage. Each input image pixel is uniquely addressed by its i–x–y coordinates, where x and y are the x–y coordinates of the pixel in the subimage contained in PE i.

The EGT algorithm consists of three major steps. First, the Sobel edge operator [20] is used to generate an *edge image* in which gray levels indicate the magnitude of the gradient. A figure of merit that indicates how well a given thresholded gray-level image matches edges in the edge image is then computed for every possible threshold. Finally, the maximum value of the figure-of-merit function is chosen to determine the threshold level. This is done for each subimage independently; thus, the threshold levels may differ from one subimage to the next. The complete EGT algorithm is most easily formulated as the SIMD procedure given in Fig. 4. Let the subimage SI be $M/\sqrt{N} \times M/\sqrt{N}$ and $SI(i,x,y)$ be a subimage pixel, where $0 \le x, y < M/\sqrt{N}$, $0 \le i < N$. The algorithm is performed for all of the subimages (all i) simultaneously.

Referring to Fig. 4, the first *for* statement clears the sumedge and nedge counters (to be described) for each possible threshold value. The next pair of nested *for* statements contains statements to calculate quantities associated with each pixel in the subimage. The Sobel operators, sx and sy, represent weighted pixel value differences in the x and y directions, respectively. The value $g(i,x,y)$ represents the gradient at pixel (i,x,y), and these values form the edge image. The presence of an edge is indicated by high edge image pixel values. Next, the local maximum and minimum pixel values over a 3×3 window are determined for each gray-level image pixel. Note that the same image pixels necessary for the calculation of the gradient can be re-used for the determination of the local maximum and minimum. The center pixel of the 3×3 window is an *edge point* if the threshold is greater than or equal to the local minimum and less than the local maximum. Running sums of the edge image pixels (gradient values) corresponding to edge points at each threshold (sumedge) and a count of the number of edge pixels for each threshold (nedge) are updated in the innermost *for* loop. In general, each PE performs this *for* statement using a different localmin and localmax and thus performs the statements in the loop (updates the sums) various numbers of times. This implies that each PE has the capability of maintaining its own loop index values. PEs are disabled when they finish their looping, because PEs must remain synchronized in SIMD mode. The total time to perform the innermost *for* loop is the maximum time taken by any PE.

The mean for each threshold (sumedge/nedge) is known as the figure of merit (merit) and is calculated in the final *for* statement using the accumulated sums. High figure of merit values indicate better matches between threshold-generated boundaries and the edges detected by the Sobel oper-

```
for thresh = 0 to 255 do
  sumedge(i, thresh) = nedge(i, thresh) = 0
for x = 0 to M/√N − 1 do begin
  for y = 0 to M/√N − 1 do begin
    sx(i,x,y) = ¼[(SI(i,x − 1,y − 1) + 2*SI(i,x − 1,y)
                + SI(i,x − 1,y + 1)) − (SI(i,x + 1,y − 1)
                + 2*SI(i,x + 1,y) + SI(i,x + 1,y + 1))]
    sy(i,x,y) = ¼[(SI(i,x − 1,y − 1) + 2*SI(i,x,y − 1)
                + SI(i,x + 1,y − 1)) − (SI(i,x − 1,y + 1)
                + 2*SI(i,x,y + 1) + SI(i,x + 1,y + 1))]
    g(i,x,y) = √(sx(i,x,y)² + sy(i,x,y)²)
    localmax(i)
      = max(SI(i,x − 1,y − 1), SI(i,x,y − 1), SI(i,x + 1,y − 1),
            SI(i,x − 1,y), SI(i,x,y), SI(i,x + 1,y),
            SI(i,x − 1,y + 1), SI(i,x,y + 1), SI(i,x + 1,y + 1))
    localmin(i)
      = min(SI(i,x − 1,y − 1), SI(i,x,y − 1), SI(i,x + 1,y − 1),
            SI(i,x − 1,y), SI(i,x,y), SI(i,x + 1,y),
            SI(i,x − 1,y + 1), SI(i,x,y + 1), SI(i,x + 1,y + 1))
    for thresh = localmin(i) to localmax(i) − 1 do begin
      sumedge(i, thresh) = sumedge(i, thresh) + g(i,x,y)
      nedge(i, thresh) = nedge(i, thresh) + 1
      end
  end
for thresh = 0 to 255 do
  merit(i, thresh) = sumedge(i, thresh)/nedge(i, thresh)
```

Fig. 4 EGT algorithm.

ator. The gray level associated with the maximum value of the figure-of-merit function is chosen for image segmentation.

The EGT algorithm is particularly well suited for SIMD parallelism because all pixels are processed similarly. This aids the PE-to-PE communication necessary when a PE must process pixels not in its subimage (i.e., in a neighbor PE). All PEs will simultaneously request the same pixel relative to their subimages. For example, when processing begins (with the upper left corner subimage pixel) all PEs will request (from the PE to their upper left) the pixel immediately above and to the left of their upper left corner pixel (if

this pixel is in the complete image). This transfer of data from upper left neighbors can occur for all PEs simultaneously. In the case of this algorithm, transmission delays incurred due to PE-to-PE data transfers can be overlapped with data processing to reduce total execution time. A total of $4(M/\sqrt{N} + 1)$ parallel transfers are needed for a $M \times M$ pixel image. The candidate interconnection networks of PASM can support these parallel transfers from any neighboring PE.

Since PE-to-PE communications in MIMD mode require explicit synchronization between the two processors for each data transfer, SIMD mode transfers should be used to provide each PE more efficiently with the one-pixel-deep border points of its subimage (from its neighbors). However, once each PE has all of the data it needs to perform the EGT algorithm, the calculations could proceed in MIMD mode. Although MIMD mode would make the execution of the innermost *for* loop more efficient (because no PEs would be disabled), this advantage must be weighed against the extra time involved in switching from SIMD to MIMD mode and requiring that each PE perform its own control flow operations for the outer two *for* loops. Control flow operations include initialization and incrementing of loop counters, evaluation of conditional expressions, and branching. These operations are performed by the MC in SIMD mode for the outer two loops and can be overlapped with the PE operations. The next step of the scenario is contour tracing.

5. CONTOUR TRACING

A contour tracing algorithm using MIMD parallelism and based on the one given in Tuomenoksa *et al.* [19] is summarized in this section. Initially, each PE contains a threshold value T for its subimage, which was calculated using the EGT algorithm of the previous section. The contour tracing algorithm has two phases. In Phase I, the PEs segment their subimages based on the threshold and all local contours (both closed and partial) are traced and recorded. In Phase II, the partial contours traced during Phase I are connected.

A *contour table* is constructed in each PE, containing an entry for every partial or complete contour in its subimage. Each contour table entry contains bookkeeping information such as the threshold value that generated the contour and a pointer to the the i–x–y sequence of the contour. Each PE also contains a *partial contour list*, which has an entry for each partial contour containing the i–x–y coordinates of its two end points and a pointer to its contour table entry.

In Phase I there is no PE-to-PE communication. Each PE uses its threshold level to segment its subimage. To create the segmented image for threshold T, subimage pixels that have a value greater than or equal to T are assigned a value of one; otherwise, the pixels are assigned a value of zero.

Contour tracing begins by scanning rows of the segmented image beginning with the top row. Scanning stops when a pixel with a value of one is found that has a zero-valued neighbor on both sides. This pixel is marked as the *start point* of a new contour, and its i–x–y coordinates are stored. For edge PEs, i.e., those on the edge of the $\sqrt{N} \times \sqrt{N}$ grid of PEs, no image points lie beyond the edge; thus, all points in the leftmost (or rightmost) column of the subimage of the PEs in the leftmost (or rightmost) column of the grid of PEs are potential start points. For all other left and right subimage edges, it is assumed that the pixel in the neighboring PE is one-valued so that spurious start points are not chosen. Bypassing a potential start point (e.g., a left subimage edge with a zero-valued neighbor in the PE to its left) is not a problem because (1) contours have multiple potential start points within the subimage and (2) the partial contours will be connected in Phase II regardless of the start point chosen.

The contour is first traced in a counterclockwise direction (CCW) if the start point has a one-valued point to its right and is first traced in a clockwise direction (CW) if the start point has a one-valued point to its left. If there are zeroes on both sides, the initial direction chosen does not matter. Consider the start point pixel as the center pixel of the 3×3 window in which direction 0 is east, 1 is northeast, and so on [21]. The CCW algorithm is stated as follows. Beginning with the neighboring pixel in direction five and incrementing by 1 modulo 8 to determine the next pixel, look for a pixel that has a value of one. When it is found, store the direction p of this new pixel and append its i–x–y coordinate to the contour sequence. Treat this pixel as a new center point of the 3×3 window. Then continue by looking for the next pixel in the contour beginning with the pixel in position $(p + 5)$ modulo 8. Tracing continues until the start point or a subimage boundary (point of indecision) is reached. The CW algorithm is similar, but scanning begins with the pixel in position zero and decrements by 1 modulo 8 to determine the next pixel. After a point is found, the pixel in position $(p + 3)$ modulo 8 is scanned. Horizontal edges that span a subimage are also recognized; however, they are treated as a special case because no start point would have been identified. An implicit assumption is that all contours to be traced define regions that have area. Examples of illegal contours that would not be traced are one-pixel-wide lines or isolated points.

A *point of indecision* is reached when a pixel from an adjacent subimage is needed to determine the next direction of the contour [19]. When a point of

indecision is reached, it is recorded as an *end point*, and the algorithm returns to the start point to trace the contour in the opposite direction until another point of indecision is reached. When tracing in the CW (or CCW) direction, the new contour pixels are inserted onto the front (or back) of the i–x–y sequence. Pixels in the thresholded image are marked so that the contour will not be retraced.

As an example, a 30 × 20 image is divided into six 10 × 10 subimages; each subimage is loaded into one of six PEs. The result of Phase I processing is shown in Fig. 5 where a dot indicates a one-valued pixel. Even though the entire object in PE 5 was located within the subimage, the left edge of the object was not traced in Phase I, because PE 5 could not determine whether the object continued into the next subimage. On the other hand, a closed contour was found in Phase I for the object in PE 4, because the object did not include any border pixels of the subimage.

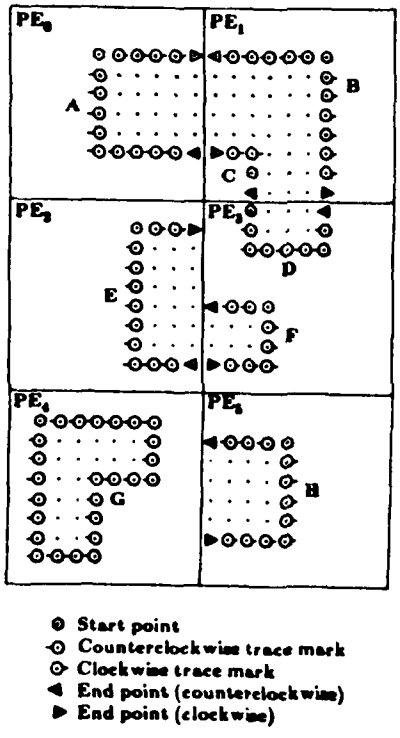


Fig. 5 Results of Phase I of contour tracing for a 30 × 20 subimage. (Based on Tuomenoksa, *et al.*, [19].)

For the example in Fig. 5, the local threshold value T is applied to the subimage in each PE. Each PE i begins scanning its respective subimage at pixel (i, 0, 0,) for a one (indicated by a dot) with a zero on either side. Depending on the start point found, tracing will proceed in either the CW or CCW direction. For example, contours A, C, E, D, and G are traced in the CCW direction first, whereas contours B, F, and H are traced in the CW direction first. In the example, PEs 1 and 3 have found two start points and have produced two traces. Once a PE has scanned the segmented image generated by its threshold, Phase I is complete.

In Phase II, each PE attempts to connect its partial contours to those located in neighboring PEs. In order for a PE to extend a contour, it must be able to access and modify contour tables that are located in other PEs. As a result, a mechanism to allow access to a contour table entry by only one PE at a time must be provided by the system and used by the contour tracing algorithm. A *semaphore* [22] associated with each contour table entry is used to indicate whether or not that entry is locked so that no other processor can access it. Semaphores are used to prevent variable access and updating problems due to interrupts. Details of these problems are beyond the scope of this paper.

For the example of Fig. 5, PE 0 might try to extend the CW end point of partial contour A by considering the possible extending pixels in PE 1 one at a time using the CW algorithm. To do this, PE 0 first locks the contour table entry for A. Then PE 0 requests that PE 1 check its partial contour lists to determine if any partial contour has the possible extending point as an end point. If such a partial contour exists, PE 1 locks the contour table entry pointed to by the partial contour list signifying that this entry is to be linked. In this case, PE 0 determines that A can be linked to B; thus, PE 1 locks B's contour table entry so that only PE 0 will be allowed to connect the partial contour. The i–x–y sequence for contour B is transferred to PE 0 and concatenated to the i–x–y sequence of partial contour A, forming a new, extended partial contour AB. If PE 0 found the contour table of partial contour B to be already locked, it will not be allowed to connect the contour. The extension of corner points is handled similarly but involves communication with more than one PE. Note that the use of semaphores prevents another PE, i.e., PE 3, from using PE 1 to access B's contour table entry which PE 1 is in the process of modifying for PE 0.

Once PE i locates a partial contour in an adjacent subimage that continues the contour and has stored the concatenated contour in its contour table, it repeats the process, if necessary, by following the contour to the next PE until the contour is closed or cannot be extended.

Independently of the actions of PE 0, PE 3 might attempt to extend contour D CCW to form the partial contour DC. If PE 3 attempted to extend

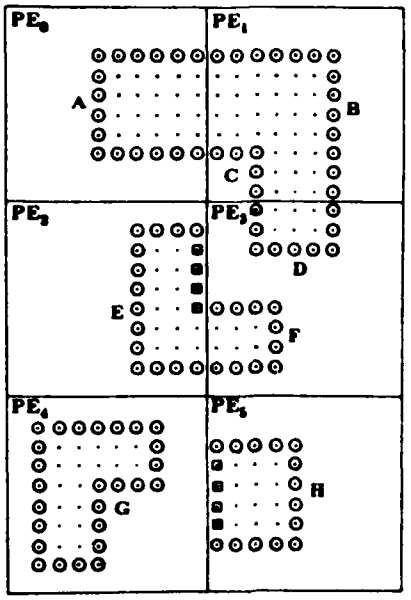


Fig. 6 Results of Phase II of contour tracing for a 30 × 20 subimage. (Based on Tuomenoksa, *et al.*, [19].)

the result, DC, when PE 0 is in the process of extending A into PE 1, it will find A locked. PE 3 then abandons its attempt to close the contour, because PE 0 is also attempting to do it, and unlocks partial contour DC. This allows PE 0 to access DC after it has appended B to A. Therefore, the closed contour ABDC is ultimately traced by PE 0. Alternatively, if PE 0 had completed linking B to A before PE 3 completed linking C to D, and PE 0 finds D locked, it would unlock AB. Thus, the closed contour would have been completely traced by PE 3. Not allowing a PE to wait for access to another PEs locked contour table entry and requiring the blocked PE to unlock its affected partial contour prevents deadlock.

Occasionally, some contour tracing operations must be performed in Phase II before certain contours can be linked. Figure 6 shows a situation in which PE 2 traces contour E along the subimage boundary in Phase II before linking it to contour F. The subimage boundary pixels of contour H are also traced in Phase II.

These examples demonstrate the basic ideas underlying the algorithms. The actual parallel algorithm details that ensure proper interaction of the PEs are complex and are not examined here.

When Phase II of the algorithm is complete, the i–x–y sequence for each contour in the image will be contained in exactly one of the PEs that contained part of the contour originally. The result of Phase II processing for the example for Fig. 5 is shown in Fig. 6. Since each PE tries to connect its contours independently, the number of the PE that finally closes a given contour is nondeterministic. Although this may not be desirable in a few cases, in general the lack of a specific protocol determining which PEs can close contours equalizes both the processing load of each PE and the number of closed contours that eventually reside in each PE.

6. ARCHITECTURAL IMPLICATIONS

The study of a parallel image processing task leads to an understanding of necessary and useful hardware features for a system such as PASM. For the example algorithms, aspects of each that have an architectural impact will be listed. Processor-specific considerations (e.g., instruction set) are also treated, because they can have a profound effect on the performance of the algorithms.

Although only two closely related algorithms were presented in the previous sections, the two could hardly have been more different in their processing demands. As discussed in Section 4, the EGT algorithm is best suited for SIMD mode. This is because the algorithm requires data that are mostly local to each PE. Also, there are approximately (or exactly) the same number of pixels to be processed in each PE, and all pixels are processed similarly.

When nonlocal data are needed in the EGT algorithm, the eight nearest-neighbor PEs comprise the set of data sources. The PE-to-PE transfer of information must be efficient, or the parallel algorithms will be slowed. In its simplest form, this communication would be handled entirely by the PEs; each PE would control the network settings (through the use of routing tags [7]) and perform all of the network protocol support (e.g., buffering, error detection). Each word transferred and each new network setting would require processor instructions. A more efficient method of PE-to-PE communication is by *direct memory access* (DMA). DMA is a means by which data can be retrieved from one memory location and stored in another without processor intervention. The DMA hardware usually operates on a cycle-stealing basis so that a PE's access to its memory is not severely affected. In its basic form, PEs in SIMD mode would enter a DMA handling routine.

This routine computes the local memory address range of the points to be transferred and sends this information to the special DMA hardware. The PEs then would compute the destination address of the PE that is to receive the data and set the network accordingly. The DMA hardware then would autonomously retrieve the information from local memory and perform the necessary network interfacing to send the data to the requesting PE. However, the PE would still be responsible for checking the incoming data (after the transfer is complete) for transmission errors and so forth. A more advanced implementation of DMA capability is the use of an intelligent *network interface unit* (NIU). Requests for data from remote PEs would be made to the local NIU, which would interpret and satisfy the request by coordinating with a remote NIU. The NIU would combine DMA capability with network protocol support. VLSI technology may allow ready fabrication of sophisticated NIUs.

As discussed in Section 4, $M/\sqrt{N}$ pixels come from each of four neighbors. For the sake of example, let $M = 4096$, $N = 1024$, and $M/\sqrt{N} = 128$. Rather than involving the *source* and *destination* PEs in the individual transfers of these points, one of the DMA modes just described would be of great use. If pixels were stored in PEs by row (rows numbered 0–127), and the transfer from PE i to PE $i + \sqrt{N}$ was selected, the DMA hardware of PE i would be instructed to transfer 128 pixels starting at the address of row 127 of the image. The DMA hardware associated with PE $i + \sqrt{N}$ would be set to read 128 pixels from the network and store them beginning at an address representing row -1. When data are transferred from a PE i to PE $i + 1$, the situation is more complicated in that image data to be transferred are not contiguous. Conventional DMA hardware only supports physical block transfers of data. Here, a strong case for an intelligient NIU is made: the NIU could accept more complicated instructions such as "transfer 128 pixels starting at address X, taking every 128th pixel."

The processing requirements (instruction set) for the EGT algorithm are not out of the ordinary. LSI technology already allows the fabrication of complete microprocessors having all required arithmetic and data manipulation operations on a single chip. Recent designs (e.g., Motorola 68000 [23]) handle a variety of data formats including bit, byte, 16-bit word, and 32-bit long word types. Floating point and special arithmetic function (e.g., square root, trigonometric) capability abounds in the form of *coprocessor* chips. Although the EGT algorithm involved only one special function (square root) in the calculation of the gradient, other algorithms such as image rotation, parallel root finding, and FFTs for speech processing make heavy use of special functions. Since many of the special arithmetic functions are calculated by iterative procedures, a strong case is made for including hardware to perform these operations rather than performing them in software. Software

procedures in which the number of iterations required is data-dependent are especially troublesome in SIMD mode, because processors must be disabled as they complete the desired number of iterations. Also, the total time to perform an operation is the maximum time required by any processor (because the PEs are synchronized). There is a slight advantage to having special-purpose arithmetic functions on the same standard CPU chip in that data to be processed need not be moved between the two devices. VLSI technology should make such combined CPU-specialized arithmetic processor chips a reality.

The processors must be capable of operating in SIMD mode efficiently. Although designs for using off-the-shelf microprocessors as SIMD/MIMD processing elements have been developed, some external hardware would be required to enable, disable, and synchronize PEs and get them to operate in slave mode, i.e., to accept instructions broadcast by a control unit rather than to take the instructions from their local memories [8]. This external hardware could be easily incorporated into a VLSI chip.

The EGT algorithm has been simulated for N ranging from 16 to 256 and a total image size of 64×64 pixels. A special-purpose SIMD simulator developed to evaluate the MC68000-based PASM design described in Kuehn *et al.* [8] was used to perform the simulations. Although the details of the simulation results are not presented here, the general trends of the results will be described.

As the number of PEs (N) decreased, the subimage size increased because a fixed-size total image of 64×64 pixels was used. For large subimages, the ratio of subimage edge pixels to total subimage pixels is low, making processing very efficient. This is because inter-PE transfers make up only a very small fraction of the total processing time. A *speedup factor* (serial execution time/parallel execution time) approaching N was obtained for arithmetic operations for this case. (A speedup of N is optimal.) As N was increased to 256 PEs, the subimage size decreased to 4×4. Here, the ratio of subimage edge pixels to total subimage pixels is very high, and inter-PE transfers make up a large percentage of the total processing time. Although the total processing time is minimized as N increases, the speedup factor decreases. The simulations imply that N should be as large as possible for the EGT algorithm to minimize the processing time. However, this would make contour tracing (the next algorithm of the scenario) inefficient, because few contours would be traced in Phase I, and heavy use of inter-PE communication would be needed to close the contours in Phase II. Thus, the scenario must be considered as a whole rather than as a sequence of individual algorithms.

Turning now to the contour tracing algorithm of Section 5 we note that both phases of the algorithm are suited to MIMD mode, because they involve

data-dependent execution times. Phase I of contour tracing requires only local data, whereas Phase II makes heavy use of nonlocal data. Phase I imposes no extraordinary requirements on the system, because there are no special arithmetic operations and no network transfers to be done. Phase II, however, with its arbitrary one-to-one connections (when transferring partial contour information between nonadjacent PEs), use of semaphores, and special signaling protocols imposes many new architectural requirements.

The interconnection network and any DMA or NIU hardware would be heavily used in Phase II processing when PEs extending partial contours probe remote PE memories that may contain the extensions of the partial contours. As in the EGT algorithm, NIU hardware would be of great use, because it could process queries about possible extensions to partial contours without interrupting the remote PE. There would be a combination of short and long messages between PEs during this phase. A short message would occur when a PE, extending a partial contour, requests information about possible extending pixels from a remote PE. If a connecting partial contour is found, a long message, consisting of the i–x–y sequence of the partial contour, would be sent. Thus the interconnection network should support a variety of message sizes so that the efficiency of sending either type of message is high.

Since semaphores play a large part in ensuring correct linking of partial contours in Phase II, processors must be equipped with *test-and-set* or similar operations to facilitate a correct semaphore implementation. Most modern microprocessors already have some semaphore capabilities.

If the system is to support the execution of the two example algorithms well, it must be capable of dynamically switching between SIMD and MIMD operation, as PASM can. With only SIMD capability, the contour tracing algorithm would be executed with huge inefficiencies, because there would be varying numbers and lengths of contours and arbitrary one-to-one communication patterns. A machine having only MIMD mode would be less seriously affected but would lengthen execution time for the EGT algorithm, due to the need for explicit synchronization for each data transfer step and the overhead of loop counter processing which is done concurrently by the MCs in SIMD mode. Thus, the capability to dynamically switch between SIMD and MIMD modes is important so that each algorithm can be executed in the most appropriate mode of parallelism.

Since PASM is an SIMD/MIMD system, the interconnection networks proposed for PASM would be capable of operating both synchronously and asynchronously. The proposed networks are of the multistage type and can perform both the nearest-neighbor and arbitrary one-to-one connections.

The design of a multi-microprocessor system that could be used as a building block for PASM is discussed in Kuehn *et al.*, [8]. This design uses

the Motorola MC68000 as the heart of both the PE and MC components. The extra hardware needed for SIMD/MIMD mode processing and communication was described. It was found that most of the extra hardware was involved in the enabling/disabling, synchronization, and instruction broadcasting for SIMD mode and in getting the PEs to switch from SIMD to MIMD mode and back again efficiently. The design highlights are described:

MC CPU. The MC CPU is a Motorola MC68000-series processor.

Fetch unit. This unit fetches instructions from MC memory in SIMD mode, determines whether they are control (MC) or data processing (PE) instructions, and broadcasts them either to the MC CPU or PE CPUs. Each instruction word in the MC memory is tagged to allow the fetch unit to determine its type. The tags are generated at assembly time.

Masking operations unit. This is specialized hardware, under the control of the MC CPU, that produces a *mask* (pattern) used to selectively enable or disable PEs (used in SIMD mode).

MC/PE interface. This is specialized hardware to queue PE instructions and enable signals broadcast to the PEs. The queue has been shown to increase the amount of program overlap between the MC and PEs. This interface is for SIMD mode; there would also be a MC/PE communication bus for MIMD mode and error-handling messages (which is not discussed here).

PE CPU. The PE CPU is a Motorola MC68000-series processor.

SIMD/MIMD mode switching logic. This is a specialized address decoder that generates instruction requests to the MC/PE interface in SIMD mode and causes local PE memory to be accessed in MIMD mode.

Network interface unit. This unit handles DMA and network protocol.

VLSI technology should be used to combine the components listed above only when some speed or complexity advantage is gained. For example, the PE CPU and SIMD/MIMD mode switching logic should be combined into a single component so that the PEs can operate equally well in SIMD and MIMD mode. This action would result in very little additional silicon area and at most a few additional pins being used. Taking this one step further, one could also fabricate the DMA and NIU hardware on the PE CPU chip. However, to allow communication on the CPU data bus (with, for example, the local memory chips) and the NIU-interconnection network bus to occur simultaneously, pins for a complete NIU bus interface would have to be added. The technology at implementation time would determine the maximum pin count and thus the suitability of this scheme.

Similarly, the MC CPU and fetch unit should be combined on one chip so that MC operations such as fetching SIMD instructions and branching are done by the same unit. The masking operations unit could easily be made a part of the MC CPU since it is not too complex; however, the number of CPU

pins would have to increase by N/Q. For the PASM design goal of $N = 1024$ and $Q = 32$, $N/Q = 32$. Again, the desirability of integrating this unit is dependent on pin count limitations. The MC/PE interface is also a candidate for inclusion on the MC chip. It would not require much silicon area, but its pin requirements are high. Since the interface queues both enable signals and instruction words to be broadcast to the PEs, an additional $N/Q + 16$ bits would be required on the MC CPU package (for MC68000 16-bit words). Thus, assuming that the number of pins that the MC CPU alone requires is P, if the masking operations is integrated with the CPU, $P + N/Q$ pins would be required; if, in addition, the MC/PE interface is integrated, $P + N/Q + 16$ pins would be required (the masking operations unit output pins to the MC/PE interface would now be internal to the chip). As has been discussed in McMillen and Siegel [24], VLSI implementation of interconnection network functions is most promising, both from a functional standpoint and a design standpoint due to network regularity.

In summary, based on our prototype plans and the expected execution needs of the contour extraction task and other image and speech processing algorithms, certain desirable system architecture features have been identified. These include dynamically switchable SIMD/MIMD capability, support for PE-to-PE communications using DMA and intelligent network interfaces, and special arithmetic function hardware. These requirements are consistent with the capabilities of a VLSI implementation of PASM.

7. SUMMARY

Contour extraction has been used as an image processing scenario to explore the advantages and implications of using the PASM parallel processing system. Use of these parallel algorithms leads to several advantages, notably speedup. Analysis of the algorithms has motivated the inclusion of several important architectural features. These features were used to discuss possible configurations of a custom-designed VLSI processor chip set for PASM. The use of algorithm characteristics to drive the design of PASM leads to a machine with features that provide the necessary flexibility for executing image and speech processing algorithms.

REFERENCES

[1] Lewis, L., Amitai, Z., and Silverman, H. F. (1983). The APS-II processor for speech recognition. *1983 IEEE Int'l. Conf. Acoustics, Speech, and Signal Processing*, 483–486.

[2] Hockney, R. W., and Jeshope, C. R. (1981). *Parallel Computers*. Adam Hilger Ltd., Bristol, England.

[3] Flynn, M. J. (1966). Very high-speed computing systems. *Proc. IEEE*, **54**, 1901–1909.

[4] Siegel, H. J., Siegel, L. J, Kemmerer, F. C., Mueller Jr., P. T., Smalley Jr., H. E., and Smith, S. D. (1981). PASM: A partitionable SIMD/MIMD system for image processing and pattern recognition. *IEEE Trans. Comp.*, **C-30**, 934–947.

[5] Kapur, R. N., Premkumar, U. V., and Lipovski, G. J. (1980). Organization of the TRAC processor-memory subsystem. *AFIPS 1980 Nat'l. Comp. Conf.*, 623–629.

[6] Sejnowski, M. C., Upchurch, E. T., Kapur, R. N., Charlu, D. P. S., and Lipovski, G. J. (1980). An overview of the Texas reconfigurable array computer. *AFIPS 1980 Nat'l. Comp. Conf.*, 631–641.

[7] Siegel, H. J. (1984). *Interconnection Networks for Large-Scale Parallel Processing: Theory and Case Studies*. Lexington Books, Lexington, Massachusetts.

[8] Kuehn, J. T., Siegel, H. J., and Hallenbeck, P. D. (1982). Design and simulation of an MC68000-based multimicroprocessor system. *1982 Int'l. Conf. Parallel Processing*, 353–362.

[9] Siegel, H. J. (1983). The PASM system and parallel image processing. In *Computer Architectures for Spatially Distributed Data* (H. Freeman and G. G. Pieroni, eds.), Springer-Verlag, New York.

[10] Mueller Jr., P. T., Siegel, L. J., and Siegel, H. J. (1980). Parallel algorithms for the two-dimensional FFT. *Fifth Int'l. Conf. Pattern Recognition*, 497–502.

[11] Siegel, L. J. (1981). Image processing on a partitionable SIMD machine. In *Languages and Architectures for Image Processing* (M. Duff and S. Levialdi, eds.) Academic Press, London, pp. 293–300.

[12] Krygiel, A. J. (1976). An implementation of the Hadamard transform on the STARAN associative array processor. *1976 Int'l. Conf. Parallel Processing*, **34**.

[13] Siegel, L. J., Siegel, H. J., and Feather, A. E. (1982). Parallel processing approaches to image correlation. *IEEE Trans. Comp.*, C-31, 208–218.

[14] Warpenburg, M. R., and Siegel, L. J. (1982). SIMD image resampling. *IEEE Trans. Comp.*, C-31, 934–942.

[15] Stone, H. S. (1971). Parallel processing with the perfect shuffle. *IEEE Trans. Comp.*, C-20, 153–161.

[16] Siegel, L. J., Siegel, H. J., Safranek, R. J., and Yoder, M. A. (1980). SIMD algorithms to perform linear predictive coding for speech processing applications. *1980 Int'l. Conf. Parallel Processing*, 193–196.

[17] Yoder, M. A., and Siegel, L. J. (1982). Dynamic time warping algorithms for SIMD machines and VLSI processor arrays. *1982 Int'l. Conf. Acoustics, Speech, and Signal Processing*, 1274–1277.

[18] Mitchell, O. R., Reeves, A. P., and Fu, K-S. (1981). Shape and texture measurements for automated cartography. *1981 IEEE Comp. Soc. Conference on Pattern Recognition and Image Processing*, 367.

[19] Tuomenoksa, D. L., Adams III, G. B., Siegel, H. J., and Mitchell, O. R.

(1983). A parallel algorithm for contour extraction: advantages and architectural implications. *1983 IEEE Comp. Soc. Symp. Computer Vision and Pattern Recognition*, 336–344.

[20] Duda, R. O., and Hart, P. E. (1973). *Pattern Classification and Scene Analysis*. John Wiley and Sons, New York.

[21] Freeman, H. (1961). Techniques for the digital computer analysis of chain-encoded arbitrary plane curves. *Proc. NEC*, 17, 421–432.

[22] Dijkstra, E. W. (1968). Cooperating sequential processes. In *Programming Languages* (F. Genuys, ed.). Academic Press, New York, pp. 43–112.

[23] Motorola Semiconductor Products Inc. (1979). *MC68000 16-bit Microprocessor User's Manual*, Motorola IC Division, Austin, Texas.

[24] McMillen, R. J., and Siegel, H. J. (1982). A comparison of cube type and data manipulator type networks. *Third Int'l. Conf. Distributed Computing Systems*, 614–621.

Paper 5

Parallel Computation of Normalized Fourier Descriptors

From the *Proceedings of the Twenty-second Annual Allerton Conference on Communication, Control, and Computing,* University of Illinois-Urbana, Monticello, Illinois, October 1984.

PARALLEL COMPUTATION OF NORMALIZED FOURIER DESCRIPTORS

KIRK D. SMITH and LEAH H. JAMIESON
School of Electrical Engineering
Purdue University
West Lafayette, IN 47907

ABSTRACT

Normalized Fourier descriptors provide an effective but computationally intensive method for performing object identification and tracking. Parallel algorithms to compute normalized Fourier descriptors are presented. The task includes sub-algorithms for conversion of chain code inputs to X-Y coordinates, filtering, resampling, Fourier transform, and normalization. MIMD and SIMD formulations are considered. The algorithms are analyzed with respect to computational complexity and communications requirements. For typical problem sizes and appropriate choice of machine size P, speedups of O(P) are achieved.

1. INTRODUCTION

Image processing algorithms are growing more complex as research is conducted. Performance demands are also increasing steadily. The major factor fueling these advances is increased speed of computer hardware. Practically, speed of computing hardware has some limitations. In the future, gains in speed may not be as dramatic. Even today, certain tasks cannot be performed because of computational bottlenecks and real-time requirements.

Clearly, a solution to these problems lies in the replication of available computing hardware. A challenging part of this field lies in the development of parallel algorithms for varied image processing tasks. In this paper, an implementation of an image processing algorithm is presented. It is representative of a wide class of image processing algorithms since it is composed of several sub-algorithms, each with different characteristics.

2. ALGORITHM OVERVIEW

Given the contour of an object in a two dimensional plane as input, a series of frequency domain coefficients which describes the image is computed. These are the Fourier descriptors, which are further processed in a normalization procedure so that they can be compared to a library of these descriptors. Output of the algorithm is the identification of the object as well as a reasonable estimate of its orientation in space [9]. The algorithm has been proven effective in identifying and tracking aircraft in flight [10].

Input to the program consists of the chain code representation of the contour of an image in a two dimensional plane. In practice, chain code inputs typically contain from 64 to 2048 points. This chain code input is then converted to X-Y coordinates of the image. After optional smoothing, the image is resampled at equally spaced intervals on the contour. Then a complex Fourier transform is performed on the resampled points. This produces a Fourier descriptor (FD).

The second logical division of the algorithm normalizes this descriptor. The goal is to scale and orient the contour by rule such that the FD from an unknown contour will always normalize to the correct library representation. Different normalizations have been proposed. Wallace's algorithm [9] is investigated here.

This material is based on work supported by the U.S. Army Research Office under Contract DAAG29-82-K-0101.

The normalization is accomplished as follows: The N most significant complex coefficients of the FD (N is typically 32), are denoted as A(−N/2 + 1) through A(N/2). This frequency domain representation of the contour is normalized by removing information relating to the relative position of the contour, its size, its starting point, and its orientation. This is accomplished by three steps:

Step 1 Set A(0) = 0.
This removes all "DC" positional information.

Step 2 Divide A(i) by $|A(1)|$, $-N/2 + 1 < i < N/2$
By definition, A(1) will be the largest coefficient, so this normalizes the size of the image such that $A(i) < 1$, for all i.

Step 3 Multiply the A(i)'s by $e^{j[(i-k)u + (1-i)v]/(k-1)}$
k is the coefficient with second largest magnitude (after A(1))
u and v are the phases of A(1) and A(k) respectively.
This simultaneous application of the rotation and starting point shift operations finds one of the normalizations satisfying $u = v = 0$. This places a major axis of the contour along the X-axis.

If $k = 2$, this normalization is unique. Otherwise the phase and starting point of the normalization must be shifted to account for the $|k - 1| - 1$ other possible normalizations. Then the correct normalization must be chosen based on some other criteria. The criterion examined here chooses the correct normalization as the one which maximizes

$$\sum_{i=-N/2+1}^{N/2} \mathrm{Re}[A(i)]\,|\,\mathrm{Re}[A(i)]|$$

A parallel implementation for the complete FD algorithm will be presented. The algorithm is divided into discinct tasks, each of which is examined individually. To achieve further parallelism, the tasks could be pipelined to increase throughput for real-time applications.

3. MACHINE MODELS

Two models of asynchronous parallel processing and one model of synchronous parallel processing will be used in the algorithms. The asynchronous models will be MIMD (Multiple Instruction Stream - Multiple Data Stream) machines; the synchronous model will be an SIMD (Single Instruction Stream - Multiple Data Stream) machine [2].

The organization assumed for an SIMD machine will be a set of P processing elements (PEs), each a processor with its own memory; a control unit; and an interconnection network. The control unit broadcasts instructions to all PEs, and each active PE executes the instruction on the data in its own memory. The interconnection network allows data to be transferred among the PEs. Examples of this model are MPP (Massively Parallel Processor) [1] and Siegel's PArtitionable SIMD/MIMD (PASM) system [5]. An MIMD machine will be assumed to consist of P processors, M memories and an interconnection network. Each processor can execute an independent instruction stream. In the Shared Memory MIMD model, the interconnection network is used to allow all processors access to all of memory. Examples of this model include the NYU Ultracomputer [4] and C.mmp [3]. In the Private Memory MIMD model, there is no global store, each processor has a local memory (M = P), and the interconnection network provides communications among separate processors. An example of this is PASM [5].

Since communication is a critical part of parallel algorithms, the types of communications needed by each of the algorithms will be analyzed. This will be a function of the way in which the data is distributed among the processors/memories. However, for a given data allocation, the precise communications requirements can be obtained for SIMD and Private Memory MIMD systems. These will be expressed in terms of a few common interconnection functions. In SIMD mode, the transfer will occur simultaneously for all active processors p, $0 \le p < P$; in a Private Memory

MIMD machine, the transfer will be a request from one PE to obtain data from another PE's memory. (In SIMD algorithms it more common to consider transferring data *from* a given PE *to* another PE. In MIMD algorithms, it may be more natural to consider a data access as a transfer of data *from* a non-local memory *to* a given PE. Since the interconnection functions we are using are symmetric, this distinction will not matter.) The interconnection functions needed will be:

(1) the class of *shift* functions, where shift ±d transfers data from PE p to PE (p ±d) mod P.

(2) the class of *cube* functions, where if $r = \log_2 P$ and $p_{r-1} \cdots p_i \cdots p_0$ is the binary representation of p, $0 \le p < P$, then

$$\text{cube}_i(p) = p_{r-1} \cdots \overline{p_i} \cdots p_0$$

where $\overline{p_i}$ is the complement of p_i.

4. DECOMPOSITION INTO PARALLEL ALGORITHMS

In this section, parallel algorithms are described for each of the subtasks required for generating normalized Fourier descriptors. The algorithms are for conversion from chain code to X-Y coordinates, filtering, resampling, Fourier transform calculation, and FD normalization.

4.1 Input Conversion

It will be assumed that the contour of the image is entered in chain code representation. Figure 1a shows a typical representation for an 8 nearest neighbor chain code. The location of point p_i is dependent upon the points p_0 through p_{i-1}. The horizontal and vertical segments (0,2,4,6) have length 1; the diagonal segments (1,3,5,7) have length $\sqrt{2}$.

An example 25-point contour with its chain code representation is given in Figure 1b. Chain code inputs of practical use contain from a few hundred to a few thousand points. This is a variable number which depends on the relative size, shape, and perspective of the object being identified. The number of chain code inputs will be assumed to be C.

Chain code input is inherently serial since each input is merely an offset from the previous input. Two parallel algorithms for this normally serial task will be described. Initially assume that $P = \sqrt{C}$. Thus, the C inputs can be divided among P PEs and each PE will be responsible for $\sqrt{C}$ chain code inputs. This can be illustrated by forming the input logically into a two dimensional array. The array of input points will be denoted as CC[n,0..C−1]. Figure 2 shows the division of the contour into segments and arranges each segment into a line of the array. We can then define parallel operations in which each PE acts on a row or column of this array.

The first parallel algorithm uses the Shared Memory MIMD model. Initially each row of the input is processed by a separate PE. This is equivalent to dividing the contour into P contiguous segments, with each PE responsible for one segment. Each PE can then assume that it has the "first" segment of the contour and assign the coordinates (0,0) to the first point. It can then compute the X-Y coordinates of the rest of its points starting from this reference. Given $\sqrt{C}$ chain code inputs in each segment, each PE assumes the first point, then generates X-Y coordinates for $\sqrt{C}$ additional points. Thus, the last point generated in PE p corresponds to the first point for PE p+1 (the point previously assumed to be (0,0)). X-Y coordinates for all the input points have now been generated, however each row of the square (each segment of the contour) has a different origin in the X-Y plane. Now a correction step is employed. Denote the X-Y coordinates of input point i as XY(i), $0 \le i < C$. Since the origin is arbitrary, set it at the point XY(0), that is XY(0) ≡ (0,0). Then the previous step correctly computed the coordinates of XY(0)

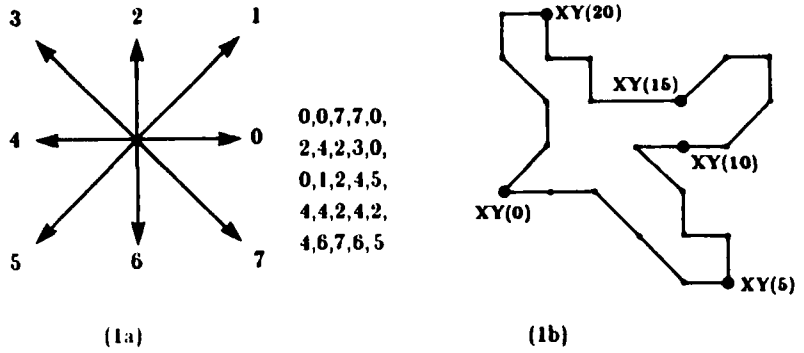


Fig. 1a. Typical chain code representation
Fig. 1b. Example 25-point chain code and contour

CCIn(0)	CCIn(1)	CCIn(2)	···	CCIn(P−1)
CCIn(P)	CCIn(P+1)			·
CCIn(2P)				·
·				·
·				·
·				·
CCIn((P−1)P)	·	·	·	CCIn(P^2−1)

Fig. 2. Division of chain code points

through XY(P). To correct the coordinates of XY(P) through XY(2P) in the second segment, we must add to each of these the coordinates of XY(P) computed in the first segment. Subject to memory access constraints, these P+1 corrections can be done concurrently. Then to correct points XY(2P) through XY(3P) in the third segment we must add the (newly corrected) XY(2P) from the second segment. All of the segment S can be corrected in parallel, however, segment S must be corrected before segment S+1, for $1 \leq S < P-1$. This correction must be done in order, for each row of the square.

This algorithm can easily be generalized to any number of input points by assigning $\lceil C/P \rceil$ consecutive points to each of the first P − 1 processors, and $C-(P-1)\lceil C/P \rceil$ points to the last processor. Some efficiency will be lost if all processors do not contain the same number of points.

In order to estimate the amount of computation performed, some assumptions about the number and types of statements will be made. Initially, synchronization overhead and memory conflicts will not be considered. The basic operations performed in the parallel algorithm are the conversion from chain code input to X-Y coordinates based on an arbitrary origin and the correction of the X-Y coordinates to the correct origin. Assume that the functions CtoX() and CtoY() convert one chain code input to the proper increment in the X or Y direction. This can be done with a simple case statement or a conversion table in which each of the 8 possible chain code inputs maps to the appropriate X and Y increments. Then the coordinates for

XY(i+1) can be obtained from XY(i) and the i-th chain code input by the pair of statements:

$$X(i+1) \leftarrow X(i) + CtoX(CCIn(i))$$

$$Y(i+1) \leftarrow Y(i) + CtoY(CCIn(i))$$

This can be considered to be 2 additions and assignments using real arithmetic or one addition and assignment using complex arithmetic. The serial algorithm consists of C calls to the conversion functions, C complex additions, and C complex assignments. The parallel approach executes in the time for $\sqrt{C}$ calls to the conversion functions, $\sqrt{C}$ assignments, and $\sqrt{C}-1$ complex additions for the initial conversion, plus $\sqrt{C}-1$ complex additions and assignments for the correction. Assuming the dominant operation is the additions, the speedup on computations is approximately

$$S \simeq \frac{C}{2\sqrt{C}-1} \simeq \frac{C}{2\sqrt{C}} = \frac{1}{2}\sqrt{C}\,.$$

For $P = \sqrt{C}$, $S \simeq P/2$.

Consider the memory references required in the above algorithm. If the data is viewed as a matrix with P data points on a side, each processor operates on a row and then a column of that matrix. In a parallel system with global memory, the store is typically divided into several memory units. Optimum efficiency comes about when each processor is accessing a different memory unit during a given memory cycle, since each memory unit can deliver only one word per memory cycle. An obvious way to distribute the data is to put each segment of the contour (row of the matrix) in a separate memory unit. During the first half of the conversion, each processor acts on a row, so the memory system operates with ideal efficiency. During the second half of the conversion, every processor acts on the same row simultaneously. This creates a large bottleneck at the memory unit containing that row. Kuck discusses this problem in [Kuc77] and suggests skewed storage techniques that eliminate these bottlenecks at the cost of more complex address computations in array accesses. There is an overhead involved in every array access, thus reducing the speedup.

In the SIMD or Private Memory MIMD model, it is assumed that accesses to the local memory can occur without contention from other processors. Consider rewriting the algorithm to use only local memories. The initial step of the algorithm is unchanged: each PE obtains X-Y coordinates for one segment of the contour, assuming an arbitrary origin. Then recursive doubling is done to produce correction values for all the segments of the contour at once. Recursive doubling is a method of computing accumulated sums across processors [8]. To show this in a program segment, assume a call to rec_dbl(val) uses the value *val* and takes care of all the communications to perform the recursive doubling. If val(i) refers to the value of *val* in PE i, then for all PEs p, $0 \le p < P$, rec_dbl(val(p)) will return the partial sum:

$$\text{rec\_dbl(val(p))} = \sum_{i=0}^{p} \text{val(i)}$$

An example of recursive doubling is shown in Figure 3.

For the Private Memory algorithm, the restriction that $P = \sqrt{C}$ is relaxed. The only assumption made is that there are D input points in each processor's local memory in the array CCIn(0..D−1). This algorithm stores a segment of the contour in each PE's local memory. Initially, each PE computes coordinates based on the assumption that its first point is at the origin (0,0). Knowing the relative coordinates of the last point in each segment, the absolute coordinates of the beginning of each segment can be computed as follows. The correction for PE 1 is given by the coordinates of the last point in PE 0; the correction for PE 2 is given by the sum of the coordinates of the last points in PEs 0 and 1; in general, the correction for PE p is the sum of the coordinates of the last points in PEs 0 through p − 1. Recursive doubling is used to compute all the needed sums simultaneously.

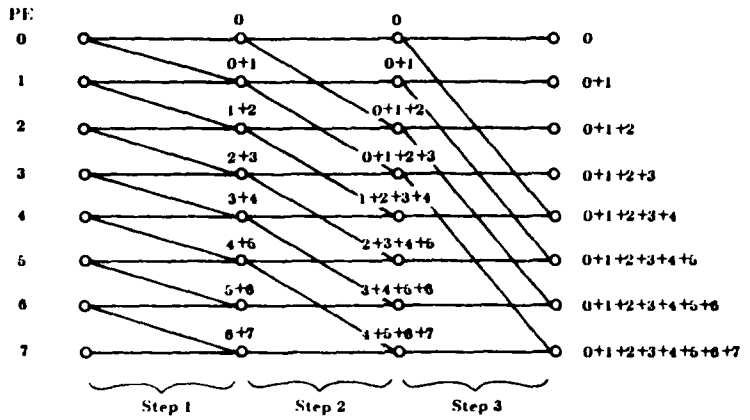


Fig. 3. Recursive doubling example for 8 PEs

Once each PE has the absolute coordinates for its first point, it can correct the rest of its segment locally. This step will be done concurrently in all PEs. The algorithm is given in Figure 4.

```
/* Local variable definitions */
P               /* Number of processing elements */
D               /* Number of data points in each processor */
CCIn(0..D-1)    /* Input chain code for one contour segment */
X(0..D)         /* X coordinates for this contour segment */
Y(0..D)         /* Y coordinates for this contour segment */
sumx            /* Partial sum of all X coordinates */
sumy            /* Partial sum of all Y coordinates */

X(0)←Y(0)←0
sumx←sumy←0

/* Compute X-Y coordinates for all points */
FOR i←0 THROUGH D-1 DO
BEGIN
        X(i+1)←X(i)+CtoX(CCIn(i))
        Y(i+1)←Y(i)+CtoY(CCIn(i))
END

/* Compute correction factors in parallel */
sumx←rec_dbl(X(D)) /* log2P transfer steps */
sumx←sumx-X(D) /* Only consider offset from previous segments */
sumy←rec_dbl(Y(D)) /* log2P transfer steps */
sumy←sumy-Y(D) /* Only consider offset from previous segments */

/* Correct each segment locally */
FOR i←1 THROUGH D-1 DO
BEGIN
        X(i)←X(i)+sumx
        Y(i)←Y(i)+sumy
END
```

Fig. 4. Input conversion algorithm for a Private Memory MIMD machine

Here the number of input points is $C = P*D$. The overall computational complexity is proportional to $D + \log_2 P$. If the assumption is kept that $C = P^2$, then $D = P = \sqrt{C}$, and the complexity is $\sqrt{C} + \log_2 \sqrt{C}$, compared with a complexity proportional to C for the serial algorithm. The operations in the first part of the algorithm (i.e., the local chain code to X-Y coordinate conversions) are the same as those used in the serial algorithm, but are performed in C/P steps instead of C. The remainder of the parallel algorithm is all overhead. The recursive doubling requires $\log_2 P$ complex additions and assignments. It also requires $\log_2 P$ points of synchronization. The local correction step takes time proportional to C/P; however, each operation is simply a complex addition, which takes less time than the original chain code to X-Y coordinate conversion step. The time is therefore dominated by the original conversion step. An O(P) speedup is expected; accounting for the extra steps, an actual speedup of P/2 is conservative.

Summarizing, two algorithms for the input conversion have been presented. Both methods are fairly regular and could be done on an SIMD machine. The first method is well suited for a Shared Memory MIMD machine, and the second method works well with either MIMD machine model. The first method effectively uses broadcasts (by placing values in memory), while the second method uses shift $+2^i$ functions, $0 \le i < \log_2 P$, for the recursive doubling. Consider representing the complexity of the algorithm as being proportional to $\alpha C/P + \log_2 P$. The actual choice of P will in general be made based on speed constraints of the application and the range of values of C. We would like to estimate P_{max}, the largest "reasonable" value for P. As an arbitrary measure, if we say that we want the cost of the computation to dominate (i.e., $\alpha C/P > \log_2 P$) and let $\alpha = 2$, then for small contours ($C = 64$), $P_{max} = 16$, and for large contours ($C = 2048$), $P_{max} = 256$.

4.2 Filtering

The filtering of the image is an optional step to remove some of the quantization noise. Typically this is a smoothing operation, in which each point is replaced by the (possibly weighted) average of itself plus W neighboring points. This can be done easily in parallel by giving each processor a section of the contour. Given a filtering window width W, each processor will need to access $(W/2)-1$ points from each adjoining section. This could be accomplished by at most W transfer steps. If a memory system is used where accesses to adjacent memories are allowed, it is important that "wrap-around" can occur. That is, PE P−1 should be a "neighbor" to PE 0. For more discussion of the filtering problem in general, see [7].

Overall, in this portion of the algorithm speedups on the order of P can be expected for small values of W. For large W, the number of accesses to data in adjacent processors may be significant. Then, properties of the parallel system will have a greater effect on the total processing time. These properties include methods of memory accesses and interconnection between processors/memory.

In the filtering step, only shift ±1 communication is needed. The number of usable PEs is related to the number of points per PE and the width W. Speed constraints may dictate how many points must be filtered by each PE. Speedup can be increased by increasing P. On the other hand, for large W, the relative effect of the transfers can be reduced by decreasing P and thus increasing C/P. Since filtering is a regular operation, it could be done easily on SIMD as well as MIMD machines.

4.3 Resampling

The input outline needs to be resampled since the Fourier descriptor algorithm requires equal distances between input samples. From chain code input, the diagonal segments are longer by a factor of $\sqrt{2}$.

The basic approach to resampling is to compute the length of the entire C-point contour, then resample it to R evenly spaced points. The length within each PE can

be computed with a speedup of P and the partial and total sums across the PEs can be computed in $\log_2 P$ steps using recursive doubling. After the total length has been obtained, the contour is divided into P groups of points such that all groups have equal length. Each PE will compute the resampled points for its own group. In the conversion and filtering stages, each PE held the same number of points. Here the PEs hold equal length groups of points, and the number of original points between two groups may differ by as much as $\sqrt{2}$. Once the boundary locations between groups have been determined, contour points may have to be moved between adjacent PEs to achieve the division into equal length groups. Once the appropriate points are collected into a PE, it can compute the resampled points for its own group. Since each PE's group has the same length, each PE will compute the same number of resampled points.

During this resampling, each processor operates primarily on local data. The only need for non-local data occurs at the ends of the contour segments. The amount of non-local data required depends on the resampling technique employed. Since most data is local, memory access is not a problem. During the recursive doubling, the interconnection network will be used. Thus, any architecture in which the communications facilities can easily support nearest neighbor (shift ± 1) and recursive doubling (shift $+2^i$) transfers should run this algorithm well.

Although SIMD machines can be used for resampling [11], MIMD execution is more suitable here because of the possible irregularities in the distances between the original samples. Either MIMD model should perform well. Again, P_{max} is chosen so that the number of points in each PE is large enough so that the amount of work is significant compared to the parallel overhead. The range of P_{max} as a function of C will be approximately the same as for the input conversion algorithm.

4.4 Fourier Transform

The FD is obtained by computing the first 32 points of the DFT on the R-point resampled contour. Here an FFT algorithm utilizes the PEs well. Since the number of contour points may be as large as 2048, but only 32 frequency domain coefficients are required for the FD, it may seem that a DFT, computing only the 32 coefficients needed, could provide similar speedups. Unfortunately, the DFT suffers from the need to broadcast all R points to all PEs, and does not approach the low computational cost of the FFT for the range of R of interest.

Using the parallel FFT algorithms in [6], a radix-2 R-point FFT can be computed in P processors (P a power of 2) in $\frac{R}{2P}\log_2 R$ complex multiplication steps, $\frac{R}{P}\log_2 R$ complex addition steps, and $\frac{R}{2P}\log_2 P$ transfer steps. If, for example, $P = 32$, then each PE will hold $R/32$ input samples, and the execution is dominated by $\frac{R}{64}\log_2 R$ complex multiplications and $\frac{R}{32}\log_2 R$ complex additions. In addition, $\frac{5}{64}R$ transfer steps are needed. These transfer steps represent the overhead of parallel execution and could account for execution overhead near the time required for the multiplication and addition steps. Even so, for $R \gg P$, the speedups are no worse than $P/2$, thus the asymptotic speedup for this portion of the algorithm is $O(P)$. In order to accomplish these gains, a communications facility is required to transfer the data at each point of synchronization. Because of the high degree of synchronization required, the FFT is best suited for SIMD rather than MIMD implementation. The data transfers are cube$_i$ functions, $0 \le i < \log_2 P$, and are done frequently. An MIMD machine of either type would be slowed by the large amount of communication and synchronization.

For an R-point radix-2 point FFT, as many as R/2 PEs could be used. However, it will be practical to use the same number of PEs as were used in the previous algorithm (resampling), so that data reallocation is not needed.

4.5 Descriptor Normalization

The remaining step normalizes the coefficients by rule so that they can be compared to a library of contour coefficients. The number of coefficients is N which is typically 32. Suppose P = N = 32. To normalize the coefficients, A(0) is set to 0 and all values are scaled by $|A(1)|$. This requires one broadcast and one parallel division. To find which coefficient is largest, the magnitudes can be computed in parallel, then the comparison can be performed in $\log_2 P$ (=5) transfers and comparisons using recursive doubling. The speedup would be on the order of only $P/\log_2 P = 6.2$ for this small section. Then depending on parameters of A(0) and A(k), the starting point and origin are shifted appropriately. This is done once if k = 2, otherwise it is done $|k - 1|$ times. Speedups can be estimated from the operations involved in shifting the origin or starting points. Either of these can be computed easily by multiplying each coefficient by a complex factor. This factor is the same across all processors for the origin adjustment and it is computed individually for the starting point adjustment. No communication or synchronization is needed, so any MIMD system should handle these well. The speedups then will be $S \simeq P = 32$ for these shifting operations.

When more than one of these normalizations are done, the "correct" normalization is computed as the one with the maximum sum

$$\sum_{i=\,N/2\,+\,1}^{N/2} \mathrm{Re}[A(i)]\,\big|\,\mathrm{Re}[A(i)]\big|$$

These terms can be computed in parallel with optimal speedup. The sum can then be formed in $\log_2 N$ steps and compared on a single processor to each of the other normalizations.

Instead of dealing with a few hundred or a few thousand data points of varying number, this procedure deals with only N ($N \simeq 32$) data points. Thus, care must be observed in estimating speedups since synchronization overhead may be a significant factor in execution speed. For this part of the processing a SIMD system may be more efficient. MIMD systems would need to have efficient synchronization mechanisms to perform well. Overall, however, a somewhat lesser speedup in this section will not significantly affect the execution time, since the number of data items has dropped from several hundred to 32, and the complexity of the operations performed in this step is not high. Since each PE could contain as few as one point each, P_{max} could be 32.

5. CONCLUSIONS

The use of a parallel machine could speed up the normalized Fourier descriptor algorithm significantly. For practical contours, the number of PEs that can be effectively used is approximately 16 or 32. The types of transfers required are the cube_i, shift ± 1, and shift $+2^i$ functions, for $0 \le i < \log_2 P$. Some sub-algorithms are better suited to MIMD architectures, others to SIMD architectures. Together, the collection of algorithms that comprise the FD task demonstrates a variety of techniques in parallel processing and shows that substantial speedups can be achieved using parallelism.

ACKNOWLEDGMENTS

The authors thank O. R. Mitchell for his help in understanding the essence of the Normalized Fourier Descriptor algorithm.

REFERENCES

[1] K. E. Batcher, "MPP -- A Massively Parallel Processor," *1979 Int'l Conf. Parallel Processing*, Aug. 1979 p. 249.

[2] M. J. Flynn, "Very High-Speed Computing Systems," *Proceedings of the IEEE*, Vol. 54, 1966, pp. 1901-1909.

[3] S. H. Fuller, R. J. Swan, W. A. Wulf, "The Instrumentation of C.mmp, a Multi-(mini)processor," *COMPCON 73 Digest of Papers*, Mar. 1973, pp. 173-176.

[4] A. Gottlieb, et al., "The NYU Ultracomputer -- Designing an MIMD Shared Memory Parallel Computer," *IEEE Trans. Comp.*, Vol. C-32, Feb. 1983, pp. 175-189.

[5] H. J. Siegel, et al., "PASM: A Partitionable SIMD/MIMD System for Image Processing and Pattern Recognition," *IEEE Trans. Comp.*, Vol. C-30, Dec. 1981, pp. 934-947.

[6] L. J. Siegel, P. T. Mueller, Jr., H. J. Siegel, "FFT Algorithms for SIMD Machines," *Seventh Annual Allerton Conf. on Communication, Control, and Computing*, Oct. 1970, pp. 1006-1014.

[7] L. J. Siegel, H. J. Siegel, A. E. Feather, "Parallel Processing Approaches to Image Correlation," *IEEE Trans. Comp.*, Vol. C-31, Mar. 1982, pp. 208-218.

[8] H. S. Stone, "Parallel Computers," in *Introduction to Computer Architecture*, H. S. Stone, ed., Science Research Associates, Chicago, 1980, pp. 372-403.

[9] T. P. Wallace and O. R. Mitchell, "Local and Global Shape Description of Two and Three-Dimensional Objects," School Elec. Eng., Purdue Univ., West Lafayette, IN, Tech. Rep. TR-EE 79-43, Sept. 1979.

[10] T. P. Wallace and O. R. Mitchell, "Analysis of Three-Dimensional Movement Using Fourier Descriptors," *IEEE Trans. Pattern Analysis and Machine Intelligence*, Vol. PAMI-2, Nov. 1980, pp. 583-588.

[11] M. R. Warpenburg and L. J. Siegel, "SIMD Image Resampling," *IEEE Trans. Comp.*, Vol. C-31, Oct. 1982, pp. 934-942.

Paper 6

Theoretical Modeling and Analysis of Special Purpose Interconnection Networks

From the Proceedings of The 4th International Conference on Distributed Computing Systems, May, 1984, pp. 256-265.

THEORETICAL MODELING AND ANALYSIS OF SPECIAL PURPOSE INTERCONNECTION NETWORKS

Robert R. Seban
Howard Jay Siegel

School of Electrical Engineering
Purdue University
West Lafayette, IN 47907, USA

Abstract

Most research in interconnection network analysis has been based on topologically regular (uniformly structured) networks. As hardware becomes less expensive, more and more distributed algorithms will be implemented by special purpose multiprocessor systems. In this paper, a formal graph/algebraic model of special purpose (topologically regular and irregular) networks is presented. These analysis techniques can be used for (a) system emulation; (b) fault tolerance; and (c) partitioning of systems.

I. Introduction

Most research in interconnection network analysis has been based on topologically regular interconnection networks such as the ILLIAC [3], Shuffle [11], multistage Cube [1], single stage Cube [19], STARAN [2], ADM [13], Mesh [14], and PM2I [18]. As hardware becomes less expensive, more and more distributed algorithms will be embedded into special purpose multiprocessor systems [15, 16, 17]. A system informally consists of a set of devices, an interconnection network, and a rule which defines the usage of the network. A device will be assumed to have two ports: one input and one output. A typical device might be a processor/memory pair, a processor only, or a memory only. Distributed algorithms for multiprocessors may give rise to special purpose irregular interconnection networks. Some effective modeling and analytical methods to study these irregular networks are needed.

This research was supported by the U.S. Army Research Office, Department of the Army, under contract number DAAG29-82-K-0101.

Problems that will benefit from precise analytical methods include:

(1) the use of system A to emulate system B (three different degrees of strictness of emulation are discussed);
(2) fault tolerance/reliability (achieved by multiple mappings of the same problem into a system); and
(3) partitioning of a system.

Some work has been done on these problems for regular interconnection networks, for example for (1)"quotient networks" [5] and for (3) partitioning theory [20, 21].

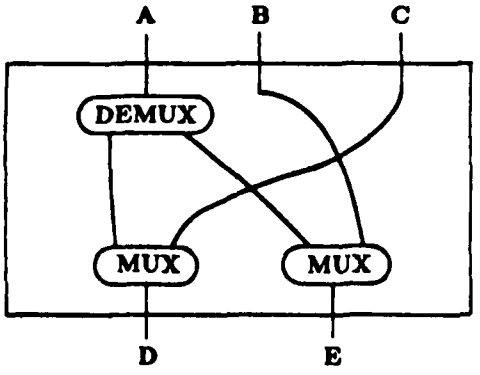
The methods developed here will allow a well defined comparison between topologies of systems. For example if system A is related to system B, and system B is related to system C, then it may be possible to say something about the relationship of system A to system C. The similarity measures are of three basic types, with each one stricter than the previous one.

The material is presented as follows: after each major definition or theorem a brief example of its application is given. In this paper it will be assumed that the reader is familiar with basic graph theory [4, 9] and basic abstract algebra [8, 10].

In section II some basic concepts are defined. The model of interconnection networks to be used is given in section III. In section IV the definitions of a system and three types of subsystems are presented and their properties analyzed. The concept of a "quasimorphism" is explained in section V. Its usage in analyzing the emulation and other problems is exemplified. Finally, in section VI, the global conclusions of this paper are discussed.

II. Basic Definitions

In this section, basic definitions needed as background for the rest of the paper are given. A general model of an interconnection network is shown in Fig. 1.



$C = \{C_0, C_1\}$
$C_0 = \{(A,D),(B,E)\}$
$C_1 = \{(A,E),(C,D)\}$
$V_I = \{A,B,C\}$, $V_O = \{D,E\}$

Fig. 1. General model of an interconnection network.

Definition 2.1:

Let V_I be the set of input labels of a network, and let V_O be the set of output labels of a network such that:

$V_I \cap V_O = \emptyset$, $V_I \neq \emptyset$, $V_O \neq \emptyset$, where $\emptyset$ is the *empty set.*

Then $C_m \subseteq V_I \times V_O \triangleq \{(v_a,v_b) \mid v_a \in V_I, v_b \in V_O\}$ is called the *I/O correspondence* of V_I with V_O. (Physically, C_m represents one state of a reconfigurable network).

Definition 2.2:

Let $C_m \subseteq V_I \times V_O$ be an I/O correspondence, then $S(C_m) \triangleq \{v_a \mid (v_a,v_b) \in C_m\}$ is called the *source set of* C_m.

Definition 2.3:

Let $C_m \subseteq V_I \times V_O$ be an I/O correspondence, then $D(C_m) \triangleq \{v_b \mid (v_a,v_b) \in C_m\}$ is called the *destination set of* C_m.

Definition 2.4:

Let $\{C_m\}$ be a set of I/O correspondences, then $S(\{C_m\}) \triangleq \bigcup_m S(C_m)$.

Definition 2.5:

Let $\{C_n\}$ be a set of I/O correspondences, then $D(\{C_n\}) \triangleq \bigcup_n D(C_n)$.

Definition 2.6:

Let $C_m \subseteq V_I \times V_O$ be an I/O correspondence.

If $v_b \neq v_d \ \forall (v_a,v_b), (v_c,v_d) \in C_m$ then the *correspondence has the property of nondestructivity.*

Definition 2.7:

Let A be a set, then $P(A) \triangleq \{S \mid S \subseteq A\}$ is the *power set.*

Definition 2.8:

Let θ be a map from A to B. Let $E \subseteq A$ then $\theta(E) \triangleq \{b \in B \mid \theta(a) = b, a \in E\}$ is the *image of* E *under* θ.

III. Interconnection Network Model

In this section, a formal graph/algebraic model of an interconnection network is presented. This model will be used to define a system in section IV.

Graph models for analyzing networks have been used by other researchers. For example, in [6, 7, 12, 22] they are used to analyze Banyan networks, and in [5] they are used to study the partitioning of regular networks. The model presented here differs from [6, 7, 12, 22] and [5] by being completely general so that it can be used to describe an arbitrary (including topologically irregular) interconnection network.

Definition 3.1:

I/O representation of network.

(1) V_I – set of input vertices
(2) V_O – set of output vertices
(3) C – set of I/O correspondences C_m, where $C_m \subseteq V_I \times V_O$
(4) $\forall C_m \in C$, where C_m has the property of nondestructivity
(5) $S(C) = V_I$
(6) $D(C) = V_O$

"K" will be used to denote a network.

Notation: $K = (C) = (\{C_m\}) = (\{\{(v_a,v_b)\}\})$

(the notation $\{(v_a,v_b)\}$ indicates a set of one or more pairs of vertices).

Physical implications: $(v_a,v_b) \in C_m$ represents network moving data from input v_a to output v_b when the state of the network is C_m. C represents the set of all possible states of the reconfigurable network.

IV. Systems and Subsystems

In this section, formal graph/algebraic definitions of a system and three types of subsystems are discussed. Also shown are basic properties of the three types of subsystems. Some theorems about subsystems are presented and brief examples of their applications are given.

The mathematical definition of system given in this section can be used to model the following object. It can be interpreted as a parallel computer system, where the vertex $v_a \in V_I$ corresponds to a device output, $v_b \in V_O$ corresponds to a device input and C_m to a state of a physical network.

Definition 4.1:

C_F – *feedback correspondence.* Let $K = (\{\{(v_a, v_b)\}\})$ be a network. If the usage of the network is such that data outputed at $v_y \in V_O$ can be fed back in $v_x \in V_I$, then $(v_x, v_y) \in C_F$.

Physical implications: This describes the situation where a processor or any other device is connected to both v_y and v_x. The device inputs data into $v_x \in V_I$ and receives data at $v_y \in V_O$. Thus if $(v_x, v_y) \in C_F$ then the same device is attached to v_x and v_y. If $(v_x, v_y) \notin C_F$ then a separate device is attached to each of v_x and v_y. Since it is assumed that each device has only one input and one output, and that a vertex can have at most one device connected to it, C_F has the following properties:

(a) if $(v_x, v_y), (v_w, v_y) \in C_F$ then $v_x = v_w$;

(b) if $(v_x, v_y), (v_x, v_z) \in C_F$ then $v_y = v_z$;

(c) $C_F \subset V_I \times V_O$.

Theorem 4.2:

C_F is map, 1:1, onto from X to Y, where $X \subseteq V_I$ and $Y \subseteq V_O$.

Proof:

Obvious by definition of C_F and properties (a), (b), (c).

□

Definition 4.3:

System. Let $K = (C) = (\{C_m\})$ be a network, with V_I and V_O, and let C_F be a feedback correspondence ($C_F \subset V_I \times V_O$), then $S = (C, C_F) = (\{C_m\}, C_F)$ is called the system.

Physical implications: The C_F precisely describes the usage of a network in a system. If $S(C_F) = V_I$ and $D(C_F) = V_O$, then the system is *fully recirculating.* If $C_F \neq \emptyset$ and either $S(C_F) \neq V_I$ or $D(C_F) \neq V_O$ (or both), then the system is *partially recirculating.* If $C_F = \emptyset$ then the system is *nonrecirculating.*

An example of a system is given in Fig. 2.

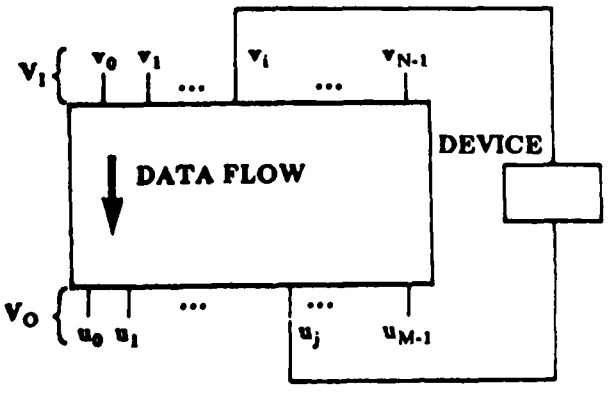


Fig. 2. Example of a system.

Definition 4.4:

Equality of systems. Let $S^{(1)} = (C^{(1)}, C_F^{(1)})$ and $S^{(2)} = (C^{(2)}, C_F^{(2)})$ be two systems. If (1) $V_I^{(1)} = V_I^{(2)}$, $V_O^{(1)} = V_O^{(2)}$; (2) $C_F^{(1)} = C_F^{(2)}$; and (3) $C^{(1)} = C^{(2)}$, then $S^{(1)}$ is equal to $S^{(2)}$.

Notation: $S^{(1)} = S^{(2)}$.

Physical implication: $S^{(1)}$ and $S^{(2)}$ are completely interchangeable.

Theorem 4.5:

Sufficiency condition for equality of systems. If (3) holds in Def. 4.4 then (1) holds.

Proof:

(a) Show: (3) $\rightarrow V_I^{(1)} = V_I^{(2)}$.
$V_I^{(1)} = S(C^{(1)}) = S(C^{(2)}) = V_I^{(2)}$.

(b) Show: (3) $\rightarrow V_O^{(1)} = V_O^{(2)}$.
$V_O^{(1)} = D(C^{(1)}) = D(C^{(2)}) = V_O^{(2)}$.

□

The implication of this theorem is that to check two systems for equality it is only necessary to examine C_F and C.

The definitions 4.6, 4.7, and 4.8 describe three different types of subsystems: a, b, and c. They are presented in order of increasing strictness.

Definition 4.6:

Subsystem type a. Let $S^{(1)} = (C^{(1)}, C_F^{(1)})$ and $S^{(2)} = (C^{(2)}, C_F^{(2)})$ be two systems.
If (1) $V_I^{(1)} \subseteq V_I^{(2)}$, $V_O^{(1)} \subseteq V_O^{(2)}$;
(2) $C_F^{(1)} \subseteq C_F^{(2)}$; and
(3) $\forall C_m^{(1)} \in C^{(1)}$
$\exists C_n^{(2)} \in C^{(2)} \ni C_m^{(1)} \subseteq C_n^{(2)} \cup C_F^{(2)}$
then $S^{(1)}$ is subsystem type a of $S^{(2)}$. ("$\ni$" means "such that")

Notation: $S^{(1)} \subseteq_a S^{(2)}$.

Example of subsystem type *a*.

Let
$S^{(2)} = (C^{(2)}, C_F^{(2)})$ be a system.
$V_I^{(2)} = \{v_0, v_1, v_2\}$
$V_O^{(2)} = \{u_0, u_1, u_2\}$
$C_F^{(2)} = \{(v_0, u_0), (v_1, u_1), (v_2, u_2)\}$
$C^{(2)} = \{C_0^{(2)}, C_1^{(2)}, C_2^{(2)}\}$
$C_0^{(2)} = \{(v_0, u_0), (v_0, u_1), (v_2, u_2)\}$
$C_1^{(2)} = \{(v_1, u_1), (v_1, u_2)\}$
$C_2^{(2)} = \{(v_2, u_1), (v_2, u_2)\}$.

Let
$V_I^{(1)} = \{v_0, v_1\}$
$V_O^{(1)} = \{u_0, u_1\}$
$C_F^{(1)} = \{(v_0, u_0), (v_1, u_1)\}$
$C^{(1)} = \{C_0^{(1)}, C_1^{(1)}\}$
$C_0^{(1)} = \{(v_0, u_0), (v_1, u_1)\}$
$C_1^{(1)} = \{(v_0, u_1)\}$.

Then (1) $(C^{(1)}, C_F^{(1)})$ is a system (denoted $S^{(1)}$).
(2) (a) $V_f^{(1)} \subseteq V_f^{(2)}$, $V_0^{(1)} \subseteq V_0^{(2)}$
(b) $C_F^{(1)} \subseteq C_F^{(2)}$
(c) $C_0^{(1)} \subseteq C_F^{(2)} \subseteq C_0^{(2)} \cup C_F^{(2)}$,
$C_f^{(1)} \subseteq C_0^{(2)} \subseteq C_0^{(2)} \cup C_F^{(2)}$
$\rightarrow S^{(1)} \subseteq_a S^{(2)}$.

Definition 4.7:

Subsystem type b. Let $S^{(1)} = (C^{(1)}, C_F^{(1)})$ and $S^{(2)} = (C^{(2)}, C_F^{(2)})$ be two systems.
If (1) $V_f^{(1)} \subseteq V_f^{(2)}$, $V_0^{(1)} \subseteq V_0^{(2)}$;
(2) $C_F^{(1)} \subseteq C_F^{(2)}$; and
(3) $\forall C_m^{(1)} \in C^{(1)} \ \exists C_n^{(2)} \in C^{(2)} \ni C_m^{(1)} \subseteq C_n^{(2)}$
then $S^{(1)}$ is subsystem type b of $S^{(2)}$.
Notation: $S^{(1)} \subseteq_b S^{(2)}$.

Example of subsystem type *b*.
Let
$S^{(2)} = (C^{(2)}, C_F^{(2)})$ be a system.
$V_f^{(2)} = \{v_0, v_1, v_2\}$
$V_0^{(2)} = \{u_0, u_1, u_2\}$
$C_F^{(2)} = \{(v_0, u_0),(v_1, u_1),(v_2, u_2)\}$
$C^{(2)} = \{C_0^{(2)}, C_1^{(2)}, C_2^{(2)}\}$
$C_0^{(2)} = \{(v_0, u_0),(v_0, u_1),(v_2, u_2)\}$
$C_1^{(2)} = \{(v_1, u_1),(v_1, u_2)\}$
$C_2^{(2)} = \{(v_2, u_1),(v_2, u_2)\}$.
Let
$V_f^{(1)} = \{v_0, v_1\}$
$V_0^{(1)} = \{u_0, u_1\}$
$C_F^{(1)} = \{(v_0, u_0),(v_1, u_1)\}$
$C^{(1)} = \{C_0^{(1)}, C_1^{(1)}\}$
$C_0^{(1)} = \{(v_0, u_0)\}$
$C_1^{(1)} = \{(v_0, u_0),(v_0, u_1)\}$.
Then (1) $(C^{(1)}, C_F^{(1)})$ is a system (denoted $S^{(1)}$).
(2) (a) $V_f^{(1)} \subseteq V_f^{(2)}$, $V_0^{(1)} \subseteq V_0^{(2)}$
(b) $C_F^{(1)} \subseteq C_F^{(2)}$
(c) $C_0^{(1)} \subseteq C_0^{(2)}$, $C_1^{(1)} \subseteq C_0^{(2)}$
$\rightarrow S^{(1)} \subseteq_b S^{(2)}$.

Definition 4.8:

Subsystem type c. Let $S^{(1)} = (C^{(1)}, C_F^{(1)})$ and $S^{(2)} = (C^{(2)}, C_F^{(2)})$ be two systems.
If (1) $V_f^{(1)} \subseteq V_f^{(2)}$, $V_0^{(1)} \subseteq V_0^{(2)}$;
(2) $C_F^{(1)} \subseteq C_F^{(2)}$; and
(3) $\forall C_m^{(1)} \in C^{(1)} \ \exists C_n^{(2)} \in C^{(2)} \ni C_m^{(1)} = C_n^{(2)}$
then $S^{(1)}$ is subsystem type c of $S^{(2)}$.
Notation: $S^{(1)} \subseteq_c S^{(2)}$.

Example of subsystem type *c*.
Let
$S^{(2)} = (C^{(2)}, C_F^{(2)})$ be a system.
$V_f^{(2)} = \{v_0, v_1, v_2\}$
$V_0^{(2)} = \{u_0, u_1, u_2\}$
$C_F^{(2)} = \{(v_0, u_0),(v_1, u_1),(v_2, u_2)\}$
$C^{(2)} = \{C_0^{(2)}, C_1^{(2)}, C_2^{(2)}\}$
$C_0^{(2)} = \{(v_0, u_0),(v_0, u_1),(v_2, u_2)\}$
$C_1^{(2)} = \{(v_1, u_1),(v_1, u_2)\}$
$C_2^{(2)} = \{(v_2, u_1),(v_2, u_2)\}$.
Let
$V_f^{(1)} = \{v_1, v_2\}$
$V_0^{(1)} = \{u_1, u_2\}$
$C_F^{(1)} = \{(v_1, u_1)\}$
$C^{(1)} = \{C_0^{(1)}, C_1^{(1)}\}$
$C_0^{(1)} = \{(v_1, u_1),(v_1, u_2)\}$
$C_1^{(1)} = \{(v_2, u_1),(v_2, u_2)\}$.
Then (1) $(C^{(1)}, C_F^{(1)})$ is a system (denoted $S^{(1)}$).
(2) (a) $V_f^{(1)} \subseteq V_f^{(2)}$, $V_0^{(1)} \subseteq V_0^{(2)}$
(b) $C_F^{(1)} \subseteq C_F^{(2)}$
(c) $C_0^{(1)} = C_1^{(2)}$, $C_1^{(1)} = C_2^{(2)}$
$\rightarrow S^{(1)} \subseteq_c S^{(2)}$.

Theorem 4.9:

Sufficiency condition for subsystem type a.
If (2) and (3) hold in Def. 4.6 then (1) holds.

Proof:

(a) Show: (2), (3) $\rightarrow V_f^{(1)} \subseteq V_f^{(2)}$.
$V_f^{(1)} = S(C^{(1)}) = S(\bigcup_m C_m^{(1)})$
$\subseteq S(\bigcup_m (C_n^{(2)} \cup C_F^{(2)}))$
since $S^{(1)} \subseteq_a S^{(2)} \rightarrow \forall C_m^{(1)} \in C^{(1)}$
$\exists C_n^{(2)} \in C^{(2)} \ni C_m^{(1)} \subseteq C_n^{(2)} \cup C_F^{(2)}$.
$S(\bigcup_m (C_n^{(2)} \cup C_F^{(2)})) \subseteq S(\bigcup_n (C_n^{(2)} \cup C_F^{(2)}))$
$\forall n \ni C_n^{(2)} \in C^{(2)}$.
$S(\bigcup_n (C_n^{(2)} \cup C_F^{(2)}))$
$= S(\bigcup_n C_n^{(2)}) \cup S(C_F^{(2)}) = V_f^{(2)}$.
Therefore $V_f^{(1)} \subseteq V_f^{(2)}$.

(b) Show: (2), (3) $\rightarrow V_0^{(1)} \subseteq V_0^{(2)}$.
Similar to (a), with $S(C^{(1)})$ and $S(C^{(2)})$ replaced by $D(C^{(1)})$ and $D(C^{(2)})$, respectively.

□

Theorem 4.10:

Sufficiency condition for subsystem type b.
If (3) holds in Def. 4.7 then (1) holds.

Proof:

Analogous to proof of Thm. 4.9 (note that (2) is not needed since C_F is not part of (3)).

□

Theorem 4.11:

Sufficiency condition for subsystem type c.
If (3) holds in Def. 4.8 then (1) holds.

Proof:

Analogous to proof of Thm. 4.10.

□

Theorem 4.12:

Let $S^{(1)} = (C^{(1)}, C_F^{(1)})$ and $S^{(2)} = (C^{(2)}, C_F^{(2)})$ be two systems.

(1) If $S^{(1)} = S^{(2)}$ then $S^{(1)} \subseteq_c S^{(2)}$.

(2) If $S^{(1)} \subseteq_c S^{(2)}$ then $S^{(1)} \subseteq_b S^{(2)}$.

(3) If $S^{(1)} \subseteq_b S^{(2)}$ then $S^{(1)} \subseteq_a S^{(2)}$.

Proof:

Obvious, follows from definitions of subsystems.

□

Theorem 4.13:

Let $S^{(1)} = (C^{(1)}, C_F^{(1)})$ and $S^{(2)} = (C^{(2)}, C_F^{(2)})$ be two systems.

If (1) $S^{(1)} \subseteq_c S^{(2)}$ and (2) $S^{(2)} \subseteq_c S^{(1)}$, then $S^{(1)} = S^{(2)}$.

Proof:

Show: (1) $V_f^{(1)} = V_f^{(2)}$, $V_O^{(1)} = V_O^{(2)}$; (2) $C_F^{(1)} = C_F^{(2)}$; and (3) $C^{(1)} = C^{(2)}$.

Remark: From Thm. 4.11 it is known that (3) $\rightarrow$ (1), so only (2) and (3) have to be shown.

(a) Show: $C_F^{(1)} = C_F^{(2)}$.

$S^{(1)} \subseteq_c S^{(2)} \rightarrow C_F^{(1)} \subseteq C_F^{(2)}$.

$S^{(2)} \subseteq_c S^{(1)} \rightarrow C_F^{(2)} \subseteq C_F^{(1)}$. $\rightarrow C_F^{(1)} = C_F^{(2)}$.

(b) Show: $C^{(1)} = C^{(2)}$.

$\forall C_m^{(1)} \in C^{(1)}$

$\exists$ unique $C_n^{(2)} \in C^{(2)} \ni C_m^{(1)} = C_n^{(2)}$.

Similarly $\forall C_q^{(2)} \in C^{(2)}$

$\exists$ unique $C_p^{(1)} \in C^{(1)} \ni C_q^{(2)} = C_p^{(1)}$.

$\rightarrow C^{(1)} = C^{(2)}$.

□

V. Quasimorphism

In this section the main results are presented. A new similarity measure between systems is defined that allows a comparison between arbitrary (regular and irregular) systems. This measure is called quasimorphism and is completely specified by two mappings called ϕ_I and ϕ_O. The quasimorphism will facilitate the analysis of following problems in parallel processing:

(a) system A emulating system B (three different degrees of strictness of emulation are discussed);

(b) fault tolerance/reliability (achieved by multiple mapping of same problem into a system);

(c) partitioning of a system.

The concept of quasimorphism provides an analytical method to study network properties that are implementation independent, such as emulation and partitioning.

Definition 5.1:

Quasimorphism type (a,b,c); where (a,b,c) means one of a or b or c.

Let $S^{(1)} = (C^{(1)}, C_F^{(1)}) = (\{C_m^{(1)}\}, C_F^{(1)})$
$= (\{\{(v_a, v_b)\}\}, \{(v_x, v_y)\})$; and
$S^{(2)} = (C^{(2)}, C_F^{(2)}) = (\{C_m^{(2)}\}, C_F^{(2)})$
$= (\{\{(w_a, w_b)\}\}, \{(w_x, w_y)\})$ be two systems.

If $\exists\ \psi = (\phi_I, \phi_O)$ such that

(1) ϕ_I: $V_f^{(1)} \rightarrow V_f^{(2)}$ is a map

(2) ϕ_O: $V_O^{(1)} \rightarrow V_O^{(2)}$ is a map

(3) ψ: $\{S\} \rightarrow \{S\}$ is a map such that:

$\psi(S^{(1)}) = \psi(\{\{(v_a, v_b)\}\}, \{(v_x, v_y)\})$

$\triangleq (\{\{(\phi_I(v_a), \phi_O(v_b))\}\}, \{(\phi_I(v_x), \phi_O(v_y))\})$

$\subseteq (a,b,c)\ S^{(2)}$,

then $\psi = (\phi_I, \phi_O)$ is *quasimorphism type (a,b,c)* from $S^{(1)}$ to $S^{(2)}$.

Physical implications of quasimorphism: Given two systems with arbitrary vertex descriptions, if there exist ψ type (a,b,c), that is, a ϕ_I and ϕ_O with the proper constraints from $S^{(1)}$ to $S^{(2)}$, then $S^{(1)}$ and $S^{(2)}$ are similar in a topological sense. The loosest similarity is ψ type a. The strictest similarity is ψ type c. The $\psi = (\phi_I, \phi_O)$ precisely describes how to handle the following problems: (1) emulation of systems; (2) identifying equivalent systems; and (3) partitioning of a network.

Additional auxiliary maps based upon ϕ_I and ϕ_O are defined to facilitate later analyses.

Definition 5.2:

ϕ_{IxO} *map.*

Let $S^{(1)} = (C^{(1)}, C_F^{(1)})$; and $S^{(2)} = (C^{(2)}, C_F^{(2)})$ be two systems.

Let ϕ_I: $V_f^{(1)} \rightarrow V_f^{(2)}$ be a map; ϕ_O: $V_O^{(1)} \rightarrow V_O^{(2)}$ be a map.

Define: ϕ_{IxO}: $V_f^{(1)} \times V_O^{(1)} \rightarrow V_f^{(2)} \times V_O^{(2)}$

$\phi_{IxO}((v_a, v_b)) \triangleq (\phi_I(v_a), \phi_O(v_b))$.

Definition 5.3:

μ *map.*

Let $S^{(1)} = (C^{(1)}, C_F^{(1)})$; and $S^{(2)} = (C^{(2)}, C_F^{(2)})$ be two systems.

Let ϕ_{IxO}: $V_f^{(1)} \times V_O^{(1)} \rightarrow V_f^{(2)} \times V_O^{(2)}$ be a map.

Define: μ: $P(V_f^{(1)} \times V_O^{(1)}) \rightarrow P(V_f^{(2)} \times V_O^{(2)})$

$\mu(\{(v_a, v_b)\}) \triangleq \{\phi_{IxO}((v_a, v_b))\}$.

Lemma 5.4:

If ϕ_I, ϕ_O are 1:1 maps then ϕ_{IxO} is 1:1 map.

Proof:

Follows from definition.

□

Lemma 5.5:

If ϕ_{IxO} is 1:1 map then μ is 1:1 map.

Proof:

Follows from definition.

□

Definition 5.6:

Alternate notation for quasimorphism.

Let $S^{(1)} = (C^{(1)}, C_F^{(1)}) = (\{\{\{(v_a,v_b)\}\}, \{(v_x,v_s)\})$ be a system.

Then: $\psi(S^{(1)}) = \psi(\{\{\{(v_a,v_b)\}\}, \{(v_x,v_s)\}\})$

$= (\{\mu(\{(v_a,v_b)\})\}, \mu(\{(v_x,v_s)\}))$

$= (\{\{\{\phi_{1:0}((v_a,v_b))\}\}\}, \{\phi_{1:0}((v_x,v_s))\})$.

Lemma 5.7:

Let $S^{(1)} = (\{\{(v_a,v_b)\}\}, \{(v_x,v_s)\})$ be a system.

Let μ: $P(V_I^{(1)} \times V_O^{(1)}) \rightarrow P(V_I^{(2)} \times V_O^{(2)})$.

Let ψ: $\{S\} \rightarrow \{S\}$ be a quasimorphism.

$\psi(S^{(1)}) = (\{\mu(\{(v_a,v_b)\})\}, \mu(\{(v_x,v_s)\}))$.

If μ is 1:1 map then ψ is 1:1 quasimorphism.

Proof:

Straightforward, but tedious.

□

Theorem 5.8:

Let $\psi = (\phi_I, \phi_O)$ be a quasimorphism.

If ϕ_I and ϕ_O are 1:1 maps then ψ is 1:1 quasimorphism.

Proof:

(1) ϕ_I, ϕ_O 1:1 $\Rightarrow$ $\phi_{I:O}$ 1:1 (Lemma 5.4).

(2) $\phi_{I:O}$ 1:1 $\Rightarrow$ μ 1:1 (Lemma 5.5).

(3) μ 1:1 $\Rightarrow$ ψ 1:1 (Lemma 5.7).

□

Physical implications: Suppose there is a $\psi^{(1)}$ such that $\psi^{(1)}(S^{(1)}) = S^{(2)}$ and $\psi^{(1)}(S^3) = S^{(2)}$ and $\psi^{(1)}$ is 1:1. First, this means that $S^{(1)} = S^{(3)}$ since $\psi^{(1)}$ is 1:1. Second, and more important from an engineering point of view, the 1:1 guarantees an efficient emulation of $S^{(1)}$ by $S^{(2)}$. That is, if all V_I were connected to processors and V_O to memories, the emulation would be such that the processing work of one processor in $S^{(1)}$ would be exactly equal to the processing work of one processor in the image of $S^{(1)}$ in $S^{(2)}$. Also, the amount of data stored in a single memory unit in $S^{(1)}$ would be exactly equal to the amount of data stored in memory unit in the image of $S^{(1)}$ in $S^{(2)}$. In other words, the mapping is regular in some sense. Analogously, the load balancing and utilization in the image of $S^{(1)}$ in $S^{(2)}$ will be identical to that in $S^{(1)}$.

Definition 5.9:

Let $\{S\}$ be a set of systems.

Define the *relation* R *of type* ψ(a,1:1) on $\{S\}$ denoted by R-ψ(a,1:1) as follows:

$(S^{(1)},S^{(2)}) \in$ R-ψ(a,1:1) iff $\exists$ a quasimorphism $\psi = (\phi_I,\phi_O)$ type a, 1:1 from $S^{(1)}$ to $S^{(2)}$.

Theorem 5.10:

Let R-ψ(a,1:1) be as in Def. 5.9 then:

(1) R-ψ(a,1:1) is reflexive,

(2) R-ψ(a,1:1) is not symmetric,

(3) R-ψ(a,1:1) is transitive.

Proof:

For (1): To show reflexivity, need to show $(S^{(1)},S^{(1)}) \in$ R-ψ(a,1:1) $\forall S^{(1)} \in \{S\}$. Let $\psi = (\phi_I,\phi_O)$ be such that ϕ_I,ϕ_O are identity maps. The rest is straightforward.

For (2): show R-ψ(a,1:1) is not symmetric.

Must show $(S^{(1)}, S^{(2)}) \in$ R-ψ(a,1:1) does not imply $(S^{(2)}, S^{(1)}) \in$ R-ψ(a,1:1).

Outline: Constructing an example of $S^{(1)}$ and $S^{(2)}$ such that $(S^{(1)},S^{(2)}) \in$ R-ψ(a,1:1) and $(S^{(2)}, S^{(1)}) \notin$ R-ψ(a,1:1). Although an example where $|V_I^{(1)}| \neq |V_I^{(2)}|$ would suffice a more interesting example is given.

Let

$S^{(1)} = (C^{(1)},C_F^{(1)})$

$V_I^{(1)} = \{v_a,v_b\}$, $V_O^{(1)} = \{u_c,u_d\}$

$C_F^{(1)} = \{(v_a,u_c), (v_b,u_d)\}$

$C^{(1)} = \{C_0^{(1)}, C_1^{(1)}\}$

$C_0^{(1)} = \{(v_b,u_d)\}$,

$C_1^{(1)} = \{(v_a,u_c),(v_a,u_d)\}$.

Let

$S^{(2)} = (C^{(2)}, C_F^{(2)})$

$V_I^{(2)} = \{w_a,w_b\}$, $V_O^{(2)} = \{x_c,x_d\}$

$C_F^{(2)} = \{(w_a,x_c),(w_b,x_d)\}$

$C^{(2)} = \{C_0^{(2)},C_1^{(2)}\}$

$C_0^{(2)} = \{(w_b,x_c)\}$,

$C_1^{(2)} = \{(w_a,x_c),(w_a,x_d)\}$.

Then $\psi = (\phi_I,\phi_O)$ with $\phi_I(v_a) = w_a$, $\phi_I(v_b) = w_b$, $\phi_O(u_c) = x_c$, $\phi_O(u_d) = x_d$ is quasimorphism type a,1:1 from $S^{(1)}$ to $S^{(2)}$, but there does not exist quasimorphism type a,1:1 from $S^{(2)}$ to $S^{(1)}$.

For (3): show R-ψ(a,1:1) is transitive.

Must show: $(S^{(1)},S^{(2)}) \in$ R-ψ(a,1:1), $(S^{(2)},S^{(3)}) \in$ R-ψ(a,1:1) $\Rightarrow$ $(S^{(1)},S^{(3)}) \in$ R-ψ(a,1:1).

Outline: Transitivity will be shown by exhibiting ψ(a,1:1) from $S^{(1)}$ into $S^{(3)}$.

(1): Let $S^{(1)} = (C^{(1)},C_F^{(1)})$; $S^{(2)} = (C^{(2)},C_F^{(2)})$; and $S^{(3)} = (C^{(3)},C_F^{(3)})$ be three systems.

(2): $(S^{(1)},S^{(2)}) \in$ R-ψ(a,1:1)

$\Rightarrow \exists\ \phi_I^{(1)} : V_I^{(1)} \rightarrow V_I^{(2)}$, 1:1

and $\exists\ \phi_O^{(1)} : V_O^{(1)} \rightarrow V_O^{(2)}$, 1:1.

(3): (2) and Lemma 5.4

$\Rightarrow \exists\ \phi_{I:O}^{(1)} : V_I^{(1)} \times V_O^{(1)} \rightarrow V_I^{(2)} \times V_O^{(2)}$, 1:1.

(2) and Lemma 5.5

$\Rightarrow \exists\ \mu^{(1)} : P(V_I^{(1)} \times V_O^{(1)}) \rightarrow P(V_I^{(2)} \times V_O^{(2)})$, 1:1.

(4): $(S^{(2)},S^{(3)}) \in$ R-ψ(a,1:1)

$\Rightarrow \exists\ \phi_I^{(2)} : V_I^{(2)} \rightarrow V_I^{(3)}$, 1:1

and $\exists\ \phi_O^{(2)} : V_O^{(2)} \rightarrow V_O^{(3)}$, 1:1.

(5): (4) and Lemma 5.4

$\Rightarrow \exists\ \phi_{I:O}^{(2)} : V_I^{(2)} \times V_O^{(2)} \rightarrow V_I^{(3)} \times V_O^{(3)}$, 1:1.

(4) and Lemma 5.5
$\rightarrow \exists\ \mu^{(2)} : P(V_I^{(2)} \times V_O^{(2)}) \rightarrow P(V_I^{(3)} \times V_O^{(3)})$, 1:1.

(6): Define: $\phi_I = \phi_I^{(2)} \circ \phi_I^{(1)} : V_I^{(1)} \rightarrow V_I^{(3)}$. ("o" *is composition of maps*)
Clearly: ϕ_I is map, 1:1.

(7): Define: $\phi_O = \phi_O^{(2)} \circ \phi_O^{(1)} : V_O^{(1)} \rightarrow V_O^{(3)}$.
Clearly: ϕ_O is map, 1:1.

(8): Define: $\phi_{IxO} = \phi_{IxO}^{(2)} \circ \phi_{IxO}^{(1)}$:
$\phi_{IxO} : V_I^{(1)} \times V_O^{(1)} \rightarrow V_I^{(3)} \times V_O^{(3)}$.
Clearly: ϕ_{IxO} is map, 1:1.

(9): Define: $\mu = \mu^{(2)} \circ \mu^{(1)}$:
$\mu : P(V_I^{(1)} \times V_O^{(1)}) \rightarrow P(V_I^{(3)} \times V_O^{(3)})$.
Clearly: μ is map, 1:1.

(10): Claim: $\psi = (\phi_I, \phi_O)$ is quasimorphism from $S^{(1)}$ to $S^{(3)}$, type a,1:1.

(11): Show: $\mu(C_F^{(1)}) \subseteq C_F^{(3)}$.
$(S^{(1)},S^{(2)}) \in R\text{-}\psi(a,1:1) \rightarrow \mu^{(1)}(C_F^{(1)}) \subseteq C_F^{(2)}$.
$(S^{(2)},S^{(3)}) \in R\text{-}\psi(a,1:1) \rightarrow \mu^{(2)}(C_F^{(2)}) \subseteq C_F^{(3)}$.
$\rightarrow \mu^{(2)}(\mu^{(1)}(C_F^{(1)})) \subseteq C_F^{(3)}$
$\rightarrow (\mu^{(2)} \circ \mu^{(1)})(C_F^{(1)}) = \mu(C_F^{(1)}) \subseteq C_F^{(3)}$.

(12): Show: $\forall\ C_m^{(1)} \in C^{(1)}\ \ \exists\ C_q^{(3)} \in C^{(3)}$
$\ni : \mu(C_m^{(1)}) \subseteq C_q^{(3)} \cup C_F^{(3)}$.

(a): $(S^{(1)},S^{(2)}) \in R\text{-}\psi(a,1:1)$
$\rightarrow \forall\ C_m^{(1)} \in C^{(1)}\ \ \exists\ C_n^{(2)} \in C^{(2)}$
$\ni : \mu^{(1)}(C_m^{(1)}) \subseteq C_n^{(2)} \cup C_F^{(2)}$.

(b): $(S^{(2)},S^{(3)}) \in R\text{-}\psi(a,1:1)$
$\rightarrow \forall\ C_n^{(2)} \in C^{(2)}\ \ \exists\ C_q^{(3)} \in C^{(3)}$
$\ni : \mu^{(2)}(C_n^{(2)}) \subseteq C_q^{(3)} \cup C_F^{(3)}$.

(c): (a), (b) $\rightarrow \forall\ C_m^{(1)} \in C^{(1)}$
$\exists\ C_q^{(3)} \in C^{(3)}\ \ni : \mu^{(2)}(\mu^{(1)}(C_m^{(1)}))$
$\subseteq \mu^{(2)}(C_n^{(2)} \cup C_F^{(2)})$
$= \mu^{(2)}(C_n^{(2)}) \cup \mu^{(2)}(C_F^{(2)})$
$\subseteq (C_q^{(3)} \cup C_F^{(3)}) \cup \mu^{(2)}(C_F^{(2)})$.

(d): (c) and $C_F^{(3)} \supseteq \mu^{(2)}(C_F^{(2)})$
$\rightarrow \mu^{(2)}(\mu^{(1)}(C_m^{(1)})) = (\mu^{(2)} \circ \mu^{(1)})(C_m^{(1)})$
$= \mu(C_m^{(1)}) \subseteq C_q^{(3)} \cup C_F^{(3)}$.

(13): (11) and (12) $\rightarrow \psi = (\phi_I, \phi_O)$ is quasimorphism type a.

(14): (9) and Lemma 5.7 $\rightarrow \psi$ is 1:1.

(15): (13) and (14) $\rightarrow (S^{(1)},S^{(3)}) \in R\text{-}\psi(a,1:1)$.

(16): Conclusion: (15) $\rightarrow$ R-ψ(a,1:1) has the property of transitivity.

□

Physical implications: If $\psi(S^{(1)}) \subseteq$ a $S^{(2)}$ then the system $S^{(2)}$ can emulate system $S^{(1)}$. The movement of the data is accomplished (a) by using the network $\{C_m^{(2)}\}$ correspondences, and (b) by using the feedback or internal connection of the device connected to both input and output of the network.

This type of emulation always exists if the $S^{(2)}$ system is partially or fully recirculating. If the network in $S^{(2)}$ is partially or fully recirculating then $\exists\ (v_x,v_y) \in C_F^{(2)}$. Then using maps $\phi_I(v_i) = v_x$ $\forall\ v_i \in V_I^{(1)}$, $\phi_O(v_j) = v_y$ $\forall\ v_j \in V_O^{(1)}$ will satisfy the necessary conditions for quasimorphism type a. This however will result in very poor load balancing. Great improvement in load balancing optimality will result if the quasimorphism is 1:1. Then each device in $\psi(S^{(1)})$ (the image of $S^{(1)}$ under ψ) will have same amount of computation (data) as the corresponding device in $S^{(1)}$.

Definition 5.11:

Let {S} be a set of systems.

Define the ***relation*** R ***of type*** ψ(b,1:1) on {S} denoted R-ψ(b,1:1) as follows:

$(S^{(1)},S^{(2)}) \in R\text{-}\psi(b,1:1)$ iff $\exists$ a quasimorphism $\psi = (\phi_I, \phi_O)$ type b, 1:1 from $S^{(1)}$ to $S^{(2)}$.

Theorem 5.12 :

Let R-ψ(b,1:1) be as in Def. 5.11 then:

(1) R-ψ(b,1:1) is reflexive,

(2) R-ψ(b,1:1) is not symmetric,

(3) R-ψ(b,1:1) is transitive.

Proof:

For (1): Reflexivity: similar to proof of Thm. 5.10.

For (2): Show: R-ψ(b,1:1) is not symmetric.

Must show $(S^{(1)},S^{(2)}) \in R\text{-}\psi(b,1:1)$ does not imply $(S^{(2)},S^{(1)}) \in R\text{-}\psi(b,1:1)$.

Outline: Constructing an example of $S^{(1)}$ and $S^{(2)}$ such that $(S^{(1)},S^{(2)}) \in R\text{-}\psi(b,1:1)$ and $(S^{(2)},S^{(1)}) \notin R\text{-}\psi(b,1:1)$.

Let
$S^{(1)} = (C^{(1)}, C_F^{(1)})$
$V_I^{(1)} = \{v_a, v_b\}$, $V_O^{(1)} = \{u_c, u_d\}$
$C_F^{(1)} = \{(v_a, u_c)\}$
$C^{(1)} = \{C_0^{(1)}, C_1^{(1)}, C_2^{(1)}\}$
$C_0^{(1)} = \{(v_a, u_d)\}$
$C_1^{(1)} = \{(v_b, u_d)\}$
$C_2^{(1)} = \{(v_a, u_c)\}$.

Let
$S^{(2)} = (C^{(2)}, C_F^{(2)})$
$V_I^{(2)} = \{w_a, w_b\}$, $V_O^{(2)} = \{x_c, x_d\}$
$C_F^{(2)} = \{(w_a, x_c)\}$
$C^{(2)} = \{C_0^{(2)}, C_1^{(2)}\}$
$C_0^{(2)} = \{(w_a, x_d), (w_a, x_c)\}$
$C_1^{(2)} = \{(w_b, x_d)\}$.

Then $\psi = (\phi_I, \phi_O)$ with $\phi_I(v_a) = w_a$, $\phi_I(v_b) = w_b$, and $\phi_O(u_c) = x_c$, $\phi_O(u_d) = x_d$ is quasimorphism type b, 1:1 from $S^{(1)}$ to $S^{(2)}$, but there does not exist quasimorphism type b, 1:1 from $S^{(2)}$ to $S^{(1)}$.

For (3): Show: R-ψ(b,1:1) is transitive.

Must show: $(S^{(1)},S^{(2)}) \in R\text{-}\psi(b,1:1)$,

$(S^{(2)},S^{(3)}) \in R\text{-}\psi(b,1{:}1)$
$\rightarrow (S^{(1)},S^{(3)}) \in R\text{-}\psi(b,1{:}1)$.

Outline: Transitivity will be shown by exhibiting $\psi(b,1{:}1)$ from $S^{(1)}$ to $S^{(3)}$.

(1): Let $S^{(1)} = (C^{(1)},C_F^{(1)})$; $S^{(2)} = (C^{(2)},C_F^{(2)})$; and $S^{(3)} = (C^{(3)},C_F^{(3)})$ be three systems.

The proof is similar to the one of Thm. 5.10 part 3, steps 2 through 11 except replace $\psi(a,1{:}1)$ by $\psi(b,1{:}1)$.

(12): Show: $\forall C_m^{(1)} \in C^{(1)} \ \exists \ C_q^{(3)} \in C^{(3)}$
$\ni \mu(C_m^{(1)}) \subseteq C_q^{(3)}$.

(a): $(S^{(1)},S^{(2)}) \in R\text{-}\psi(b,1{:}1)$
$\rightarrow \forall C_m^{(1)} \in C^{(1)} \ \exists \ C_n^{(2)} \in C^{(2)}$
$\ni \mu^{(1)}(C_m^{(1)}) \subseteq C_n^{(2)}$.

(b): $(S^{(2)},S^{(3)}) \in R\text{-}\psi(b,1{:}1)$
$\rightarrow \forall C_n^{(2)} \in C^{(2)} \ \exists \ C_q^{(3)} \in C^{(3)}$
$\ni \mu^{(2)}(C_n^{(2)}) \subseteq C_q^{(3)}$.

(c): (a), (b)
$\rightarrow \forall C_m^{(1)} \in C^{(1)} \ \exists \ C_q^{(3)} \in C^{(3)}$
$\ni \mu^{(2)}(\mu^{(1)}(C_{in}^{(1)})) \subseteq \mu^{(2)}(C_n^{(2)}) \subseteq C_q^{(3)}$.

(13): (11) and (12) $\rightarrow \psi = (\phi_I,\phi_O)$ is quasimorphism type b.

(14): (9) and Lemma 5.7 $\rightarrow \psi$ is 1:1.

(15): (13) and (14) $\rightarrow (S^{(1)},S^{(3)}) \in R\text{-}\psi(b,1{:}1)$.

(16): *Conclusion*: (15) $\rightarrow$ R-ψ(b,1:1) has the property of transitivity.

□

Physical implication: If $\psi(S^{(1)}) \subseteq$ b $S^{(2)}$ then the system $S^{(2)}$ can emulate system $S^{(1)}$. The movement of the data is accomplished by using the network correspondences $(C_m^{(2)})$. This type of emulation is harder to achieve than the type a since the C_F contribution cannot be used to move the data. Again, as in type a, the load balancing optimality will greatly increase if the quasimorphism is 1:1. If the quasimorphism is 1:1 then the load balancing as well as utilization in the image of $S^{(1)}$ in $S^{(2)}$ will be identical to that in $S^{(1)}$.

The quasimorphism can be used to map multiple copies of system $S^{(1)}$ into $S^{(2)}$, where $\psi^{(1)}(S^{(1)}) \cap \psi^{(2)}(S^{(1)}) = \emptyset$ is necessary additional constraint. This will allow tandem crosschecking of partial results of a computation and therefore can be used as error detection for fault tolerance.

Definition 5.13:

Let {S} be a set of systems.

Define the *relation* R *of type* $\psi(c,1{:}1)$ on {S} denoted R-ψ(c,1:1) as follows:

$(S^{(1)},S^{(2)}) \in R\text{-}\psi(c,1{:}1)$ iff $\exists$ a quasimorphism $\psi = (\phi_I,\phi_O)$ type c , 1:1 from $S^{(1)}$ to $S^{(2)}$.

Theorem 5.14: Let R-ψ(c,1:1) be as in Def. 5.13 then:

(1) R-ψ(c,1:1) is reflexive,

(2) R-ψ(c,1:1) is not symmetric,

(3) R-ψ(c,1:1) is transitive.

Proof:

For (1): Reflexivity similar to proof of Thm. 5.10.

For (2): Show R-ψ(c,1:1) is not symmetric.

Must show, $(S^{(1)},S^{(2)}) \in R\text{-}\psi(c,1{:}1)$ does not imply $(S^{(2)},S^{(1)}) \in R\text{-}\psi(c,1{:}1)$.

Outline: Constructing an example of $S^{(1)}$ and $S^{(2)}$ such that $(S^{(1)},S^{(2)}) \in R\text{-}\psi(c,1{:}1)$ and $(S^{(2)},S^{(1)}) \notin R\text{-}\psi(c,1{:}1)$.

Let
$S^{(1)} = (C^{(1)}, C_F^{(1)})$
$V_I^{(1)} = \{v_a,v_b\}$, $V_O^{(1)} = \{u_c,u_d\}$
$C_F^{(1)} = \{(v_b,u_d)\}$
$C^{(1)} = \{C_0^{(1)},C_1^{(1)}\}$
$C_0^{(1)} = \{(v_a,u_d),(v_b,u_c)\}$
$C_1^{(1)} = \{(v_a,u_c)\}$.

Let
$S^{(2)} = (C^{(2)},C_F^{(2)})$
$V_I^{(2)} = \{w_a,w_b\}$, $V_O^{(2)} = \{x_c,x_d\}$
$C_F^{(2)} = \{(w_b,x_d)\}$
$C^{(2)} = \{C_0^{(2)},C_1^{(2)},C_2^{(2)}\}$
$C_0^{(2)} = \{(w_a,x_c)\}$
$C_1^{(2)} = \{(w_a,x_d),(w_b,x_c)\}$
$C_2^{(2)} = \{(w_b,x_d)\}$.

Then $\psi = (\phi_I,\phi_O)$ with $\phi_I(v_a) = w_a$, $\phi_I(v_b) = w_b$, and $\phi_O(u_c) = x_c$, $\phi_O(u_d) = x_d$ is quasimorphism type c, 1:1 from $S^{(1)}$ to $S^{(2)}$, but there does not exist quasimorphism type c, 1:1 from $S^{(2)}$ to $S^{(1)}$.

For (3): Show R-ψ(c,1:1) is transitive.

Must show: $(S^{(1)},S^{(2)}) \in R\text{-}\psi(c,1{:}1)$,
$(S^{(2)},S^{(3)}) \in R\text{-}\psi(c,1{:}1) \rightarrow (S^{(1)},S^{(3)}) \in R\text{-}\psi(c,1{:}1)$.

Outline: Transitivity will be shown by exhibiting $\psi(c,1{:}1)$ from $S^{(1)}$ into $S^{(3)}$.

(1): Let $S^{(1)} = (C^{(1)},C_F^{(1)})$; $S^{(2)} = (C^{(2)},C_F^{(2)})$; and $S^{(3)} = (C^{(3)},C_F^{(3)})$ be three systems.

The proof is similar to the one of Thm. 5.10 part 3, steps 2 through 11 except replace $\psi(a,1{:}1)$ by $\psi(c,1{:}1)$.

(12): Show $\forall C_m^{(1)} \in C^{(1)} \ \exists \ C_q^{(3)} \in C^{(3)}$
$\ni \mu(C_m^{(1)}) = C_q^{(3)}$.

(a): $(S^{(1)},S^{(2)}) \in R\text{-}\psi(c,1{:}1)$
$\rightarrow \forall C_m^{(1)} \in C^{(1)} \ \exists \ C_n^{(2)} \in C^{(2)}$
$\ni \mu^{(1)}(C_m^{(1)}) = C_n^{(2)}$.

(b): $(S^{(2)},S^{(3)}) \in R\text{-}\psi(c,1{:}1)$
$\rightarrow \forall C_n^{(2)} \in C^{(2)} \ \exists \ C_q^{(3)} \in C^{(3)}$
$\ni \mu^{(2)}(C_n^{(2)}) = C_q^{(3)}$.

(c): (a), (b)
$\rightarrow \forall C_m^{(1)} \in C^{(1)} \ \exists \ C_q^{(3)} \in C^{(3)}$
$\ni \mu^{(2)}(\mu^{(1)}(C_m^{(1)})) = \mu^{(2)}(C_n^{(2)}) = C_q^{(3)}$.

(13): (11) and (12) $\rightarrow \psi = (\phi_I, \phi_O)$ is quasimorphism type c.

(14): (9) and Lemma 5.7 $\rightarrow \psi$ is 1:1.

(15): (13) and (14) $\rightarrow (S^{(1)}, S^{(3)}) \in R\text{-}\psi(c,1{:}1)$.

(16): *Conclusion:* (15) $\rightarrow R\text{-}\psi(c,1{:}1)$ has the property of transitivity.

□

Physical implications of a quasimorphism type c. Since it is required in type b that

$$\forall C_m^{(1)} \in C^{(1)} \exists\ C_n^{(2)} \in C^{(2)} \ni C_m^{(1)} \subseteq C_n^{(2)}$$

there may be some side effects caused by $C_n^{(2)}$ emulating the correspondence $C_m^{(1)}$. Moreover, these uncontrolled side effects will not allow partitions to operate independently. That is, connections that are part of $C_n^{(2)}$, but not part of $C_m^{(1)}$, may be established when $C_n^{(2)}$ is used to emulate $C_m^{(1)}$. This may or may not be a problem. To analyze this potential problem, the type c was defined. With a type c quasimorphism, when the system $S^{(2)}$ emulates system $S^{(1)}$, the movement of the data is accomplished by a subset of $C^{(2)}$. The difference between type b and type c is that in type c,

$$\forall C_m^{(1)} \in C^{(1)}\ \exists\ C_n^{(2)} \in C^{(2)} \ni C_m^{(1)} = C_n^{(2)}.$$

This requirement will eliminate the side effects that type b has. More importantly it means that $\psi(S^{(1)})$ is actually an autonomous subsystem of $S^{(2)}$. The autonomous property will be explored further in a later paper studying partitionability.

VI. Conclusions

In this paper a theoretical basis for analyzing both topologically regular and irregular interconnection networks was developed. A rigid graph/algebraic model that can be applied to both regular and irregular interconnection networks was defined and its usefulness and flexibility was demonstrated in subsequent analyses. An important and very useful measure of similarity of networks called quasimorphism was introduced. Three types of quasimorphism were defined between two systems $S^{(1)}$ and $S^{(2)}$, where type a is the least strict and type c the most strict. Necessary conditions for each type of quasimorphism are given and their properties analyzed. The model and the quasimorphism relation provide the necessary theoretical background for studying the following problems of parallel processing.

(a) Emulation of system $S^{(1)}$ by system $S^{(2)}$.

(b) Fault tolerance method achieved by concurrent execution of multiple copies of the same problem.

(c) Partitioning of a system.

Future work includes characterizing the necessary conditions for partitioning of a system and studying multilevel quasimorphisms for analyzing systems involving multiple networks.

References

[1] G. B. Adams III and H. J. Siegel, "The extra stage cube: a fault-tolerant interconnection network for supersystems," *IEEE Trans. Comput.*, Vol. C-31, May 1982, pp. 443-454.

[2] K. E. Batcher, "STARAN parallel processor system hardware," *AFIPS Conf. Proc. 1974 Nat'l. Computer Conf.*, May 1974, pp. 405-410.

[3] W. J. Bouknight, S. A. Denneberg, D. E. McIntyre, J. M. Randall, A. H. Sameh, and D. L. Slotnick, "The Illiac IV system," *Proc. IEEE*, Vol. 60, Apr. 1972, pp. 369-388.

[4] J. A. Bondy and U. S. R. Murty, *Graph Theory with Applications*, North Holland Publishing, 1976.

[5] J. P. Fishburn and R. A. Finkel, "Quotient networks," *IEEE Trans. Comput.*, Vol. C-31, Apr. 1982, pp. 288-295.

[6] L. R. Goke, "Banyans networks for partitioning multiprocessors systems," Ph.D. Thesis, University of Florida, June 1976.

[7] L. R. Goke and G. J. Lipovski, "Banyan networks for partitioning multiprocessor systems," Symp. Computer Architecture, Dec. 1973, pp. 21-28.

[8] C. B. Hanneken, *Introduction to Abstract Algebra*, Dickenson Publishing, 1968.

[9] F. Harary, *Graph Theory*, Addison Wesley Publisher, 1969.

[10] I. N. Herstein, *Topics in Algebra*, Xerox College Publishing, 1975.

[11] T. Lang and H. S. Stone, "A shuffle-exchange network with simplified control," *IEEE Trans. Comput.*, Vol. C-25, Jan. 1976, pp. 55-66.

[12] G. J. Lipovski and M. Malek, "A Theory for Interconnection Networks," Electrical Engineering Department, University of Texas at Austin, TRAC Report 41, Oct. 1982.

[13] R. J. McMillen and H. J. Siegel, "Routing schemes for the augmented data manipulator network in an MIMD system," *IEEE Trans. Comput.*, Vol. C-31, Dec. 1982, pp. 1202-1214.

[14] D. Nassimi and S. Sahni, "An optimal routing algorithm for mesh-connected parallel computers," *Journal ACM*, Vol. 27, Jan. 1980, pp. 6-29.

[15] M. C. Pease, "An adaptation of the fast Fourier transform for parallel processing," *Journal ACM*, Vol. 15, Apr. 1968, pp. 252-264.

[16] A. P. Reeves and R. R. Seban, "The moment computer," *15th Hawaii Int'l. Conf. System Sciences*, Jan. 1982, pp. 388-396.

[17] R. R. Seban, "Parallel Computer Architecture Parallel Algorithms and the Theory of Image Analysis Using Cartesian Moments," Master's Thesis, Purdue University, Aug. 1982.

[18] R. R. Seban and H. J. Siegel, "Shuffling with the Illiac and PM2I SIMD Networks," *IEEE Trans. Comput.*, scheduled to appear in Vol. C-33, No. 5, May 1984.

[19] H. J. Siegel, "A model of SIMD machines and a comparison of various interconnection networks," *IEEE Trans. Comput.*, Vol. C-28, Dec. 1979, pp. 907-917.

[20] H. J. Siegel, "The theory underlying the partitioning of permutation networks," *IEEE Trans. Comput.*, Vol. C-29, Sept. 1980, pp. 791-801.

[21] H. J. Siegel, *Interconnection Networks for Large Scale Parallel Processing: Theory and Case Studies*, Lexington Books, Lexington, MA, 1984.

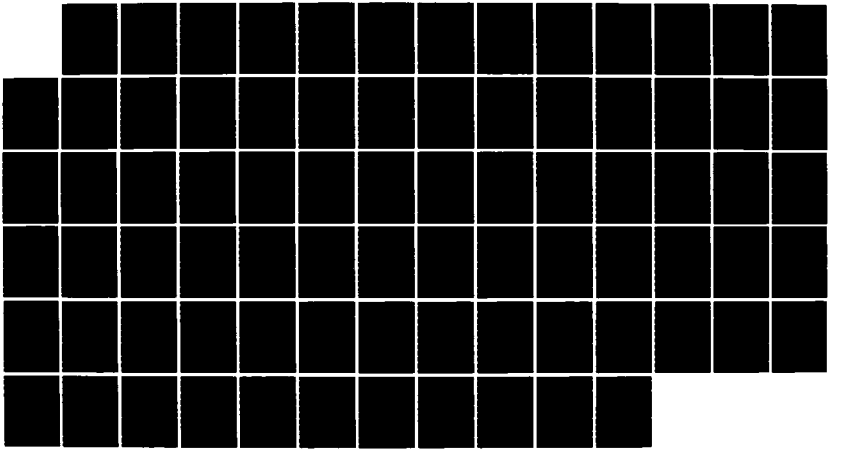
[22] P. V. Uppaluru, "A theoretical basis for analysis and partitioning of regular SW banyans," Ph.D. Thesis, The University of Texas at Austin, 1981.

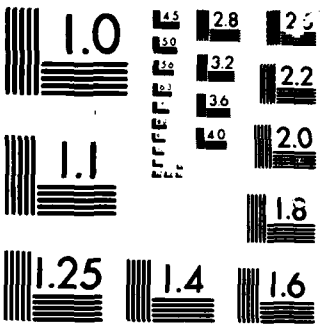
Paper 7

Analysis of Partitionability Properties of Topologically Arbitrary Interconnection Networks

AD-A167 315 DISTRIBUTED COMPUTING FOR SIGNAL PROCESSING: MODELING OF ASYNCHRONOUS PAR..(U) PURDUE UNIV LAFAYETTE IN SCHOOL OF ELECTRICAL ENGINEERING L H JAMIESON ET AL. 2/2

UNCLASSIFIED MAR 86 ARO-18790.17-EL DAAG29-82-K-0101 F/G 9/2 NL





MICROCOPY CHART

From the Proceedings of The 5th International Conference on Distributed Computing Systems, May 1985.

ANALYSIS OF PARTITIONABILITY PROPERTIES OF TOPOLOGICALLY ARBITRARY INTERCONNECTION NETWORKS

Robert R. Seban
Howard Jay Siegel

PASM Parallel Processing Laboratory
School of Electrical Engineering
Purdue University
West Lafayette, IN 47907, USA

Abstract

As hardware becomes less expensive, more and more distributed algorithms will be implemented by special purpose multiprocessor systems. An important component of such a system is the processor interconnection network. A general model of interconnections is used to formally study composition, decomposition, and partitionability properties of networks. For the reasons of implementation efficiency and reliability, these properties of networks are salient factors. Three different types of partitionability are distinguished and described and their properties shown. An algorithm is presented and proven correct that will accept as its input an arbitrary interconnection network and will produce one of four possible outputs: (1) the network is not partitionable; (2), (3), and (4) the network is partitionable in one of the three types of partitionability described.

I. Introduction

As hardware becomes less expensive, more and more distributed algorithms will be embedded into special purpose multiprocessor systems [e.g., 9, 10, 12]. Most current research on interconnection networks is specific to a single network or a single class of networks; it consists of defining a model for the network or class to be analyzed and using it for the analysis [16]. This method suffers from the following drawback: the model usually holds for only the network or class in question and therefore the analytical results are useful only for that network or class. A solution to this problem is to define a completely general model as was done [11]. By using this model, analytical results are applicable to most classes of networks. The model was defined and used in [11] to analyze the emulation properties of networks.

This research was supported by the U.S. Army Research Office, Department of the Army, under contract number DAAG29-82-K-0101, and the Purdue Research Foundation David Ross Grant 1984/85 number 0857.

In this paper the properties of network composition, decomposition, and partitionability are analyzed. An algorithm is developed which will output one of the following:

(1) The network is not partitionable.

(2) The network is partitionable into subnetworks with common control signals and the combination of the of the subnetworks will exactly generate all interconnection patterns of the original network.

(3) The network is partitionable into subnetworks with separate control signals and the combination of the subnetworks will exactly generate all interconnection patterns of the original network.

(4) The network is partitionable into subnetworks with separate control signals and the combination of the subnetworks will generate a superset of interconnection patterns of the original network.

The partitionability property of interconnection networks for parallel computer systems is important for the following reasons.

(1) If the network is partitionable then the resource allocation of only a subset of the total resources is possible. This can be used as follows.
   (a) The allocation of only a subset of the total resources is possible so that a user can use only a small part of the machine for program development and to use the whole machine when the program is developed.
   (b) In a multiple user environment the partitioning provides a natural protection among users.
   (c) In a multitasking environment the partitioning provides a protection among independent tasks.

(2) If the network is partitionable the fault tolerance of the system increases as follows.
   (a) A method of graceful degradation is possible by separating the faulty section from the correctly operating ones.

CH2149-3/85/0000/0173$01.00©1985 IEEE

(b) If in addition to being a partitionable network, the sections are isomorphic, then an increase of reliability may be realized by multiple mappings of the same task onto the multiple sections and tandem cross checking of partial results.

(c) It is possible to construct a fault tolerant network using a partitionable network as a core as will be shown in future work.

(3) If the network is partitionable, then there is an efficient implementation in terms of hardware and control. The network can be implemented as a set of network components each with its own set of inputs and outputs.

(a) If an input/output belongs to the input/output set of a component network then it does not belong to a different component, and the routing of the data paths on a VLSI chip or on a printed circuit board will be simplified.

(b) In addition, only the subset of controls that affect a particular partition will be connected to it, therefore the control lines routing may be simplified.

The results presented here are applicable to all network topologies. It is assumed here that the reader is familiar with basic graph theory [1, 6] and basic abstract algebra [5, 7]

The paper is organized as follows. In section II the basic concepts are defined. The definition of an interconnection network with an arbitrary topology is given in section III. In section IV three different types of partitionability of interconnection networks are described. In section V an algorithm is presented which determines if a network is partitionable, and if it is, differentiates among three types of partitionability.

II. Basic Definitions

In this section, basic definitions needed as background for the rest of the paper are given.

Let the set of input labels of a graph/algebraic structure be denoted by V_I and the set of output labels of the structure be denoted by V_O. All graph/algebraic structures defined in this paper over $V_I \times V_O$ will assume that $V_I \cap V_O = \emptyset$, $V_I \neq \emptyset$, $V_O \neq \emptyset$, where $\emptyset$ is the *empty set* and $V_I \times V_O = \{\langle v_a, v_b\rangle \mid v_a \in V_I, v_b \in V_O\}$.

The following notation will be used throughout this paper. The symbols are enclosed in a pair of double quotation marks.

"{","}" - delimiters for set.

"(",")" - function application and grouping of operations.

"<",">" - delimiters for n-tuple.

"[","]" - used as defined in context.

Definition 2.1: Let $C_m \subseteq [V_I \times V_O]$, then C_m is an *I/O correspondence over* $V_I \times V_O$.

Definition 2.2: Let $C_m \subseteq [V_I \times V_O]$ such that $\langle v_a, v_b\rangle, \langle v_c, v_d\rangle \in C_m \rightarrow v_b \neq v_d$, then the C_m is a *nondestructive correspondence.* (Physically, C_m represents one state of a reconfigurable network).

Definition 2.3: Let $C[V_I \times V_O] \triangleq \{C_m \subseteq [V_I \times V_O] \mid C_m$ is nondestructive$\}$. Then $C[V_I \times V_O]$ is called the *C-set over* $V_I \times V_O$.

Definition 2.4: Let $C_m \in C[V_I \times V_O]$, then $s(C_m) \triangleq \{v_a \mid \langle v_a, v_b\rangle \in C_m\}$ is the *source set of* C_m.

Definition 2.5: Let $C_m \in C[V_I \times V_O]$, then $d(C_m) \triangleq \{v_b \mid \langle v_a, v_b\rangle \in C_m\}$ is the *destination set of* C_m.

Definition 2.6: Let $C = \{C_m \mid m=1,2,...,n\} \subseteq C[V_I \times V_O]$, then $s(C) \triangleq \bigcup_m s(C_m)$ is the *source set of C.*

Definition 2.7: Let $C = \{C_m \mid m=1,2,...,n\} \subseteq C[V_I \times V_O]$, then $d(C) \triangleq \bigcup_m d(C_m)$ *is the destination set of C.*

III. Interconnection Network Model

In this section, a formal graph/algebraic model of an interconnection network is presented. Graph models for analyzing networks have been used by other researchers. For example, in [3, 4, 8, 15] they are used to analyze regular SW-banyan networks, and in [2] they are used to study the partitioning of regular networks. The model presented here differs from [3, 4, 8, 15] and [2] by being completely general so that it can be used to describe an arbitrary (including topologically irregular) interconnection network.

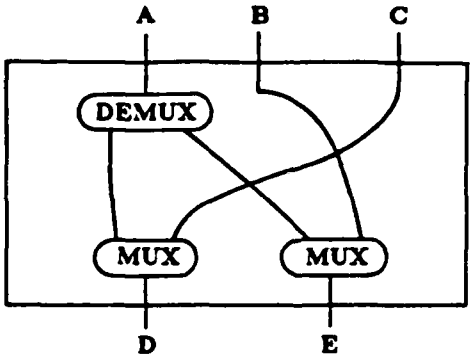
Definition 3.1: Let $K = \langle C\rangle$ be such that:

(1) $C \subseteq C[V_I \times V_O]$.
(2) $V_I = s(C)$.
(3) $V_O = d(C)$.
(4) $|C| \geq 2$.

Then, $K = \langle C\rangle$ is an *I/O representation of a reconfigurable network over* $V_I \times V_O$.

An example of an arbitrary interconnection network and description of it using this notation is shown in Fig. 1.

Physical implications: $\langle v_a, v_b\rangle \in C_m$, $C_m \in C$ represents the network moving data from input v_a to output v_b when the state of the network is C_m. C represents the set of all possible states of the reconfigurable network.



$C = \{C_0, C_1\}$

$C_0 = \{\langle A, D\rangle, \langle B, E\rangle\}$

$C_1 = \{\langle A, E\rangle, \langle C, D\rangle\}$

$V_I = \{A, B, C\}$, $V_O = \{D, E\}$

Fig. 1. Example of an arbitrary interconnection network.

Definition 3.2: Let $K[V_I \times V_O] \triangleq \{K \mid K = \langle C\rangle$ is a network over $V_I \times V_O\}$. Then $K[V_I \times V_O]$ is called the *K-set over* $V_I \times V_O$.

Definition 3.3: Let $K^1 \in K[V_I^1 \times V_O^1]$, $K^1 = \langle C^1\rangle$, and $K^2 \in K[V_I^2 \times V_O^2]$, $K^2 = \langle C^2\rangle$, be two networks such that:

(1) $V_I^1 \subseteq V_I^2$, $V_O^1 \subseteq V_O^2$.

(2) $\forall C_m^1 \in C^1\ \exists C_n^2 \in C^2\ \ni: C_m^1 \subseteq C_n^2$.

Then K^1 *is subnetwork of type b of* K^2. *Notation:* $K^1 \subseteq_b K^2$.

Definition 3.4: Let $K^1 \in K[V_I^1 \times V_O^1]$, $K^1 = \langle C^1\rangle$, and $K^2 \in K[V_I^2 \times V_O^2]$, $K^2 = \langle C^2\rangle$, be two networks such that:

(1) $V_I^1 \subseteq V_I^2$, $V_O^1 \subseteq V_O^2$.

(2) $\forall C_m^1 \in C^1\ \exists C_n^2 \in C^2\ \ni: C_m^1 = C_n^2$.

Then K^1 *is subnetwork of type c of* K^2. *Notation:* $K^1 \subseteq_c K^2$.

Note: The reason for referring to these subnetworks as types b and c is to make this notation consistent with the definitions of subsystems in [11].

Definition 3.5: Let $K^1 \in K[V_I^1 \times V_O^1]$, $K^1 = \langle C^1\rangle$, and $K^2 \in K[V_I^2 \times V_O^2]$, $K^2 = \langle C^2\rangle$, be two networks such that:

(1) $V_I^1 = V_I^2$, $V_O^1 = V_O^2$.

(2) $C^1 = C^2$.

Then K^1 *is equal to* K^2. *Notation:* $K^1 = K^2$.

Theorem 3.6: Let $K^1 \in K[V_I^1 \times V_O^1]$, $K^1 = \langle C^1\rangle$, and $K^2 \in K[V_I^2 \times V_O^2]$, $K^2 = \langle C^2\rangle$, be two networks. If $\forall C_m^1 \in C^1\ \exists C_n^2 \in C^2\ \ni: C_m^1 \subseteq C_n^2$, then $K^1 \subseteq_b K^2$.

Proof: (1): Show $V_I^1 \subseteq V_I^2$.

$(\forall C_m^1 \in C^1)(\exists C_n^2 \in C^2)(C_m^1 \subseteq C_n^2)$

$\rightarrow (\forall C_m^1 \in C^1)(C_m^1 \subseteq C_{g(m)}^2)$

$\rightarrow (\forall C_m^1 \in C^1)(s(C_m^1) \subseteq s(C_{g(m)}^2))$

$\rightarrow (\bigcup_m s(C_m^1) \subseteq \bigcup_m s(C_{g(m)}^2))$

$\rightarrow (\bigcup_m s(C_m^1) \subseteq \bigcup_n s(C_n^2)) \rightarrow V_I^1 \subseteq V_I^2$.

(2): Show $V_O^1 \subseteq V_O^2$.

Similar to (1) except replace the s set by the d set. □

Theorem 3.7: Let $K^1 \in K[V_I^1 \times V_O^1]$, $K^1 = \langle C^1\rangle$, and $K^2 \in K[V_I^2 \times V_O^2]$, $K^2 = \langle C^2\rangle$, be two networks. If $\forall C_m^1 \in C^1\ \exists C_n^2 \in C^2\ \ni: C_m^1 = C_n^2$, then $K^1 \subseteq_c K^2$.

Proof: Show $V_I^1 \subseteq V_I^2$ and $V_O^1 \subseteq V_O^2$.

The proofs are similar to proof of Theorem 3.5. □

Theorem 3.8: Let $K^1 \in K[V_I^1 \times V_O^1]$, $K^1 = \langle C^1\rangle$, and $K^2 \in K[V_I^2 \times V_O^2]$, $K^2 = \langle C^2\rangle$, be two networks. If $C^1 = C^2$ then $K^1 = K^2$.

Proof: (1): Show $V_I^1 = V_I^2$.

$C^1 = C^2 \rightarrow s(C^1) = s(C^2) \rightarrow V_I^1 = V_I^2$.

(2): Show $V_O^1 = V_O^2$.

$C^1 = C^2 \rightarrow d(C^1) = d(C^2) \rightarrow V_O^1 = V_O^2$. □

IV. Composition and Decomposition of Networks

This section describes a "horizontal" composition and decomposition of networks. The discussion here is presented for the composition of two networks into one and the decomposition of one network into two. However, it can be generalized into the composition of n networks into one and decomposition of one network into n, $n > 2$. What is meant by the *horizontal composition* of two networks K^1 and K^2 is that $V_I^1 \cap V_I^2 = \emptyset$ and $V_O^1 \cap V_O^2 = \emptyset$. Similarly, the *horizontal decomposition* of K into two networks K^1 and K^2 will result in $V_I^1 \cap V_I^2 = \emptyset$ and $V_O^1 \cap V_O^2 = \emptyset$. Two types of composition (decomposition) are described. One, the *σ-composition (decomposition)* corresponds to the physical situation where the controls of the individual subnetworks of the network are independent. The other type is the *τ-composition (decomposition)*, which corresponds to the physical situation where the controls of the individual subnetworks of the network are dependent upon one another.

This section conceptually consists of two parts. In part one the definition of the σ-composition is given and some of its basic properties are presented. In part two the definition of the τ-composition is given and its properties are described.

Definition 4.1: Let $K^1 \in K[V_I^1 \times V_O^1]$, $K^1 = <C^1>$, and $K^2 \in K[V_I^2 \times V_O^2]$, $K^2 = <C^2>$, be two networks such that: $(V_I^1 \cup V_O^1) \cap (V_I^2 \cup V_O^2) = \emptyset$. Define *$\sigma$-map* as follows: $K^1 \sigma K^2 = <C^1> \sigma <C^2> \triangleq <\{C_p^1 \cup C_r^2 \mid C_p^1 \in C^1, C_r^2 \in C^2\}>$.

This describes the composition of two networks where the controls of the two networks are independent from one another.

Lemma 4.2: Let $K^1 \in K[V_I^1 \times V_O^1]$ and $K^2 \in K[V_I^2 \times V_O^2]$ be two networks such that:
$(V_I^1 \cup V_O^1) \cap (V_I^2 \cup V_O^2) = \emptyset$.
Then $K^1 \sigma K^2 = K^2 \sigma K^1$.

Proof: Obvious from the definition of σ-map and commutativity property of set union.

□

Theorem 4.3: Let $K^1 \in K[V_I^1 \times V_O^1]$, $K^1 = <C^1>$, and $K^2 \in K[V_I^2 \times V_O^2]$, $K^2 = <C^2>$, be two networks such that: $(V_I^1 \cup V_O^1) \cap (V_I^2 \cup V_O^2) = \emptyset$. Then $K^1 \sigma K^2 \in K[(V_I^1 \cup V_I^2) \times (V_O^1 \cup V_O^2)]$.

Proof: Let $\{C_p^1 \cup C_r^2 \mid C_p^1 \in C^1, C_r^2 \in C^2\} = C^3$, $C_m^3 \in C^3$, and $C[(V_I^1 \cup V_I^2) \times (V_O^1 \cup V_O^2)] = C^*$.

(1): Show $C^3 \subseteq C^*$.
Clearly $C_m^3 \subseteq [(V_I^1 \cup V_I^2) \times (V_O^1 \cup V_O^2)]$.
Must show nondestructivity.
$<u_a,u_b>, <u_c,u_d> \in C_m^3 \rightarrow$ three cases.

(1.1): $<u_a,u_b>, <u_c,u_d> \in C_p^1$, $C_p^1 \in C^1 \rightarrow u_b \neq u_d$.

(1.2): $<u_a,u_b>, <u_c,u_d> \in C_r^2$, $C_r^2 \in C^2 \rightarrow u_b \neq u_d$.

(1.3): $<u_a,u_b> \in C_p^1$, $C_p^1 \in C^1$, and $<u_c,u_d> \in C_r^2$, $C_r^2 \in C^2$.
$(V_I^1 \cup V_O^1) \cap (V_I^2 \cup V_O^2) = \emptyset$
$\rightarrow V_O^1 \cap V_O^2 = \emptyset \rightarrow u_b \neq u_d$.

(1.4): (1.1), (1.2), and (1.3) $\rightarrow C_m^3 \in C^*$
$\rightarrow C^3 \subseteq C^*$.

(2): Show $s(C^3) = V_I^1 \cup V_I^2$.
$s(C^3) = s(\{C_p^1 \cup C_r^2 \mid C_p^1 \in C^1, C_r^2 \in C^2\})$
$= \{s(C_p^1) \cup s(C_r^2) \mid C_p^1 \in C^1, C_r^2 \in C^2\} =$
$\{s(C_p^1) \mid C_p^1 \in C^1\} \cup \{s(C_r^2) \mid C_r^2 \in C^2\} =$
$s(C^1) \cup s(C^2) = V_I^1 \cup V_I^2$.

(3): Show $d(C^3) = V_O^1 \cup V_O^2$.
Similar to 2 except replace the s set by the d set.

(4): Show $|C^3| \geq 2$.

(4.1): $|C^3| = |\{C_p^1 \cup C_r^2 \ni C_p^1 \in C^1, C_r^2 \in C^2\}|$.

(4.2): $C_p^1 \cup C_r^2 \neq C_s^1 \cup C_t^2$, $p \neq s$ or $r \neq t \rightarrow$ all C_m^3 are distinct.

(4.3): (4.1), (4.2) $\rightarrow |C^3| = |C^1| \cdot |C^2| \geq 2 \cdot 2 = 4$.

□

Lemma 4.4: Let $K^1 \in K[V_I^1 \times V_O^1]$, $K^2 \in K[V_I^2 \times V_O^2]$, and $K^3 \in K[V_I^3 \times V_O^3]$ be three networks such that $(V_I^a \cup V_O^a) \cap (V_I^b \cup V_O^b) = \emptyset$, $a \neq b$, $a,b = 1,2,3$, then $(K^1 \sigma K^2) \sigma K^3 = K^1 \sigma (K^2 \sigma K^3)$.

Proof: Obvious from the definition of σ-map and the associativity property of set union.

□

Definition 4.5: Let $K \in K[V_I \times V_O]$ be a network. Let $\{K^1, K^2, \ldots, K^n \mid K^i \in K[V_I^i \times V_O^i]\}$ be a set of networks such that: $K = K^1 \sigma K^2 \sigma \cdots K^n$. Then
(1) $K^1 \sigma K^2 \sigma \cdots K^n$ is called a *σ-decomposition of K.*
(2) $\{K^1, K^2, \ldots, K^n\}$ is called a *σ-decomposition set of K.*
(3) K^i is called a *σ-decomposition element of K.*
(4) K is the *σ-composition* of $K^1 \sigma K^2 \sigma \cdots K^n$.

Definition 4.6: Let $K \in K[V_I \times V_O]$ be a network. If the only possible σ-decomposition is $K = K^1$ then K is called a *σ-prime network.*

Definition 4.7: Let $K \in K[V_I \times V_O]$ be a network and let $K = K^1$. Then K^1 is called the *trivial σ-decomposition of K.*

Lemma 4.8: Let $K \in K[V_I \times V_O]$ be a network. Then K has a σ-decomposition.

Proof: Let $K = K^1$ be the trivial σ-decomposition of K.

□

Definition 4.9: Let $K \in K[V_I \times V_O]$ be a network. Let $K = K^1 \sigma K^2 \sigma \cdots K^n$ be a σ-composition, where $\forall j$, K^j is a σ-prime network. Then $K^1 \sigma K^2 \sigma \cdots K^n$ is called a *σ-composition prime of K.*

This decomposition can be used as a canonical form of a network. Notice that this implies $V_I = \bigcup_{i=1}^{n} V_I^i$ and $V_O = \bigcup_{i=1}^{n} V_O^i$ (where the notation $D = A \cup B$ means $D = A \cup B$ and $A \cap B = \emptyset$).

Theorem 4.10: Let $K \in K[V_I \times V_O]$, $K = <C>$, be a network. Let $K = K^1 \sigma K^2 \sigma \cdots K^n$ be any σ-decomposition. Then: $n \leq \log_2 |C|$.

Proof: Let $K^j = <C^j>$.

(1): K^j is a network $\rightarrow |C^j| \geq 2$.

(2): $|C| = \prod_{j=1}^{n} |C^j| \geq 2^n$.

(3): $n = \log_2 2^n \leq \log_2 |C|$. □

This can be used as an upper bound on number of networks in a σ-decomposition set.

Theorem 4.11: Let $K \in K[V_I \times V_O]$, $K = <C>$, be a network.

(1) If K has a nontrivial σ-decomposition then $|C|$ is not a prime number.

(2) If $|C|$ is a prime number then K does not have a nontrivial σ-decomposition.

Proof: Follows from the proof of Theorem 4.10. □

This counting principle can be used as a necessary condition on a σ-decomposition of a network.

Theorem 4.12: Let $K^1 \in K[V_I^1 \times V_O^1]$, $K^1 = <C^1>$, and $K^2 \in K[V_I^2 \times V_O^2]$, $K^2 = <C^2>$, be two networks such that: $(V_I^1 \cup V_O^1) \cap (V_I^2 \cup V_O^2) = \emptyset$. Let $K^3 = K^1 \sigma K^2$ be a σ-composition.

(1) If $\emptyset_C \in C^2$ then $K^1 \subseteq_c K^3$ where $\emptyset_C$ is the correspondence consisting of no edges, i.e., no connections between the set of inputs and the set of outputs.

(2) If $\emptyset_C \notin C^2$ then $K^1 \subseteq_b K^3$, but not $K^1 \subseteq_c K^3$.

(3) If $\emptyset_C \in C^1$ then $K^2 \subseteq_c K^3$.

(4) If $\emptyset_C \notin C^1$ then $K^2 \subseteq_b K^3$, but not $K^2 \subseteq_c K^3$.

Proof:

Case 1: Show $K^1 \subseteq_c K^3$.

(1): $K^3 = K^1 \sigma K^2 \rightarrow \forall C_m^1 \in C^1, \forall C_n^2 \in C^2$ $\exists C_p^3 \in C^3 \ni: C_p^3 = C_m^1 \cup C_n^2$.

(2): (1) and $\emptyset_C \in C^2$ $\rightarrow \forall C_m^1 \in C^1 \ \exists C_p^3 \in C^3 \ \ni: C_m^1 = C_p^3$ $\rightarrow K^1 \subseteq_c K^3$.

Case 2: Show $K^1 \subseteq_b K^3$ but not $K^1 \subseteq_c K^3$.

(1): Same as Case 1.

(2): (1) and $\emptyset_C \notin C^2 \rightarrow (\forall C_m^1 \in C^1 \exists C_p^3 \in C^3$ $\ni: C_m^1 \subset C_p^3)$ and $(\forall C_m^1 \in C^1 \ \nexists C_p^3 \in C^3 \ni$ $C_m^1 = C_p^3) \rightarrow K^1 \subseteq_b K^3$ and not $K^1 \subseteq_c K^3$.

Case 3 and 4: Same as Case 1 and 2 by the commutati ity of the σ-composition (Lemma 4.2). □

Definition 4.13: Let $K^1 \in K[V_I^1 \times V_O^1]$, $K^1 = <C^1>$, and $K^2 \in K[V_I^2 \times V_O^2]$, $K^2 = <C^2>$, be two networks such that:

(1) $(V_I^1 \cup V_O^1) \cap (V_I^2 \cup V_O^2) = \emptyset$, and

(2) $|C^1| = |C^2|$.

Define τ_α-map as follows:

(1) Define $\alpha: C^1 \rightarrow C^2$, map 1:1 and onto.

(2) $K^1 \tau_\alpha K^2 = <C^1> \tau_\alpha <C^2> \triangleq <\{C_p^1 \cup C_r^2 |$ $\alpha(C_p^1) = C_r^2, C_p^1 \in C^1, C_r^2 \in C^2\}>$.

This describes the composition of two networks where the controls are dependent in the sense that choosing a C_p^1 in C^1 means $\alpha(C_p^1)$ must be selected in C^2. Thus, the α map exactly specifies how the controls are dependent. The basic difference between the σ-map and τ_α-map is as follows. Suppose $K^1 = <C^1>$ and $K^2 = <C^2>$.

If $K^3 = K^1 \sigma K^2$, $K^3 = <C^3>$, then

(a) $|C^3| = |C^1| \cdot |C^2|$ and

(b) C_p^1 is a subset of $|C^2|$ correspondences in C^3.

If $K^3 = K^1 \tau_\alpha K^2$ then

(a) $|C^3| = |C^1| = |C^2|$ and

(b) C_p^1 is a subset of one correspondence in C^3, specifically $C_p^1 \cup \alpha(C_p^1)$.

Definition 4.14: Let $K \in K[V_I \times V_O]$ be a network. Let $\{K^1, K^2, \ldots, K^n | \ K^i \in K[V_I^i \times V_O^i]\}$ be a set of networks such that: $K = K^1 \tau_\alpha K^2 \tau_\alpha \cdots K^n$. Then

(1) $K^1 \tau_\alpha K^2 \tau_\alpha \cdots K^n$ is called a *τ-decomposition of K.*

(2) $\{K^1, K^2, \ldots, K^n\}$ is called a *τ-decomposition set of K.*

(3) K^i is called a *τ-decomposition element of K.*

(4) K is the *τ-composition* of $K^1 \tau_\alpha K^2 \tau_\alpha \cdots K^n$.

Definition 4.15: Let $K \in K[V_I \times V_O]$, $K = <C>$, be a network. *If there exist* $K^1 \in K[V_I^1 \times V_O^1]$, $K^1 = <C^1>$, and $K^2 \in K[V_I^2 \times V_O^2]$, $K^2 = <C^2>$, two networks such that: (1) $V_I^1 \cup V_I^2 = V_I$, and (2) $V_O^1 \cup V_O^2 = V_O$, then:

(1) If $K^1 \tau_\alpha K^2 = K$, then K is a *τ-partitionable* network.

(2) If $K^1 \sigma K^2 = K$, then K is a *strictly σ-partitionable* network.

(3) If $K^1 \sigma K^2 \neq K$ and $K^1 \sigma K^2 \supseteq_c K$, then K is a *$\sigma$-partitionable* network.

Note that strictly σ-partitionable implies: $|C| = |C^1| \cdot |C^2|$ and $C = \{C_s^1 \cup C_t^2 | \ C_s^1 \in C^1, C_t^2 \in C^2\}$. In contrast σ-partitionable implies: $|C| < |C^1| \cdot |C^2|$ and $C \subset \{C_s^1 \cup C_t^1 | \ C_s^1 \in C^1, C_t^2 \in C^2\}$. If K is a τ-partitionable network then it is also a σ-partitionable. It is not strictly σ-partitionable because it is strictly σ-partitionable only if $|C^1| \cdot |C^2| = |C|$ and it si τ-partitionable only if $|C^1| = |C^2| = |C|$, which implies $|C^1| = |C^2| = |C| = 1$; however, $|C^1|, |C^2|, |C| \geq 2$, by Definition 3.1. Also note that if there exists a σ-prime composition of K, then K is a strictly σ-partitionable network.

V. Partitionability Algorithm

In this section an algorithm is presented that has an input any general network (with an arbitrary topological structure) and which produces one of four possible outputs.

(1) The network is not partitionable.

(2) The network is r-partitionable.

(3) The network is strictly σ-partitionable.

(4) The network is σ-partitionable .

The engineering interpretation of the four outputs is as follows:

(1) The network is not partitionable into disjoint subnetworks.

(2) The network is partitionable into subnetworks with common control signals that are dependent upon one another and the combination of the subnetworks will exactly generate all interconnection patterns of the original network.

(3) The network is partitionable into subnetworks with independent control signals and the combination of the subnetworks will exactly generate all interconnection patterns of the original network.

(4) The network is partitionable into subnetworks with independent control signals and the combination of the subnetworks will generate a superset of interconnection patterns of the original network.

The algorithm can be programmed on a computer and if the output of the algorithm is (2) or (3) then it will produce a more efficient implementation of the network in terms of data path hardware and possibly control implementation. In case (4), even though a superset of the states of the original network is obtained, the implementation produced by the algorithm will be efficient in most instances. The following definitions are needed to discuss the algorithm and prove its correctness.

Definition 5.1: Let $K \in K[V_I \times V_O]$, $K = \langle C \rangle$. Let $C_m \in C$ and $\langle v_a, v_b \rangle \in C_m$ be an edge (directed). Denote the undirected arc associated with the directed edge of $\langle v_a, v_b \rangle$ by $\langle \overline{v_a, v_b} \rangle$. Let $G[V_I \times V_O] \triangleq \{ \langle \overline{v_a, v_b} \rangle \mid \langle v_a, v_b \rangle \in C_m, \forall C_m \in C \}$. Then $G[V_I \times V_O]$ is the ***underlying undirected graph of K.***

Definition 5.2: Let $G[V_I \times V_O]$ be the underlying undirected graph of $K \in K[V_I \times V_O]$. Then the connected subgraphs of $G[V_I \times V_O]$ are called *components of* $G[V_I \times V_O]$.

Notation: Components are denoted by $B^1, B^2, \ldots, B^n$. Denote the vertices associated with B^r by V_I^r and V_O^r, $V_I^r \subseteq V_I$, $V_O^r \subseteq V_O$. In a component B^r there exists a path from each node to every other node and there is no path between any two nodes from different components. Clearly $G[V_I \times V_O] = \bigcup_r B^r$, $\bigcup_r V_I^r = V_I$, and $\bigcup_r V_O^r = V_O$.

Definition 5.3: Let $G[V_I \times V_O]$ be the underlying graph of $K \in K[V_I \times V_O]$, $K = \langle C \rangle$. Let $C_m \in C$ and let B^r be a component of $G[V_I \times V_O]$. Define the *projection p of C_m onto B^r* as follows:
$p(C_m, B^r) \triangleq \{ \langle v_a, v_b \rangle \in C_m \mid \langle \hat{v}a, \bar{v}_b \rangle \in B^r \}$.

Lemma 5.4: Let $G[V_I \times V_O]$ be the underlying graph of $K \in K[V_I \times V_O]$, $K = \langle C \rangle$. Let $C_m \in C$ and let $\{B^1, B^2, \ldots, B^n\}$ be the set of all components of $G[V_I \times V_O]$. Then
$C_m = p(C_m, B^1) \cup p(C_m, B^2) \cup \cdots p(C_m, B^n)$.

Proof: (1): Show
$p(C_m, B^i) \cap p(C_m, B^j) \neq \emptyset \rightarrow B^i = B^j$.

(1.1): $p(C_m, B^i) \cap p(C_m, B^j) \neq \emptyset \rightarrow$
$\langle v_a, v_b \rangle \in p(C_m, B^i)$,
$\langle v_a, v_b \rangle \in p(C_m, B^j)$.

(1.2): $\langle v_a, v_b \rangle \in p(C_m, B^i) \rightarrow$
$\langle v_a, v_b \rangle \in C_m$, $\langle \overline{v_a, v_b} \rangle \in B^i$.

(1.3): $\langle v_a, v_b \rangle \in p(C_m, B^j) \rightarrow$
$\langle v_a, v_b \rangle \in C_m$, $\langle \overline{v_a, v_b} \rangle \in B^j$.

(1.4): $\langle \overline{v_a, v_b} \rangle \in B^i$, $\langle \overline{v_a, v_b} \rangle \in B^j$, and
$G[V_I \times V_O] = \bigcup_r B^r \rightarrow B^i = B^j$.

(2): Show $C_m = \bigcup_i p(C_m, B^i)$.

(2.1): Show $C_m \subseteq \bigcup_i p(C_m, B^i)$.

$\langle v_a, v_b \rangle \in C_m \rightarrow \langle \overline{v_a, v_b} \rangle \in G[V_I \times V_O]$
$\rightarrow \exists B^j, \langle \overline{v_a, v_b} \rangle \in B^i \rightarrow$
$\langle v_a, v_b \rangle \in p(C_m, B^j) \rightarrow$
$\langle v_a, v_b \rangle \in \bigcup_i p(C_m, B^i)$.

(2.2): Show $C_m \supseteq \bigcup_i p(C_m, B^i)$.

$\langle v_a, v_b \rangle \in \bigcup_i p(C_m, B^i) \rightarrow \exists B^r$,
$\langle v_a, v_b \rangle \in p(C_m, B^r) \rightarrow \langle v_a, v_b \rangle \in C_m$.

(3): (1) and (2) $\rightarrow$ $C_m = \bigcup_i p(C_m, B^i)$. □

Definition 5.5: Let $G[V_I \times V_O]$ be the underlying undirected graph of $K \in K[V_I \times V_O]$, $K = \langle C \rangle$. Let B^i be a component of $G[V_I \times V_O]$. Define the *residue set modulo B^i* as follows:
$r(B^i) \triangleq \{ p(C_b, B^i) \mid \forall C_b \in C \}$.

Theorem 5.6: Let B^r be a component of the underlying graph $G[V_I \times V_O]$ of $K \in K[V_I \times V_O]$, $K = \langle C \rangle$. Let $r(B^r)$ be the residue set modulo B^r, B^r over $V_I^r \times V_O^r$. If $|r(B^r)| \geq 2$ then $\langle r(B^r) \rangle \in K[V_I^r \times V_O^r]$. $\langle r(B^r) \rangle$ is called a ***component network of K*** denoted by $K(B^r)$.

Proof: (1): Show $\dot{C}_s \in r(B^r) \rightarrow \dot{C}_s \in C[V_I^r \times V_O^r]$.
$\dot{C}_s \in r(B^r) = \{p(C_s,B^r) \ni C_s \in C\} \rightarrow$
$\exists C_x \in C, \dot{C}_s = p(C_x,B^r) \rightarrow$
$\dot{C}_s \in C[V_I^r \times V_O^r]$.

(2): Show $s(r(B^r)) = V_I^r$.

(2.1): Show $s(\{p(C_s,B^r) \ni C_s \in C\}) \subseteq V_I^r$.
$u_s \in s(\{p(C_s,B^r) \ni C_s \in C\}) \rightarrow \exists C_b \in C$,
$<u_s,u_b> \in C_b$, $<\overline{u_s,u_b}> \in B^r \rightarrow$
$u_s \in V_I^r$.

(2.2): Show $s(\{p(C_s,B^r) \ni C_s \in C\}) \supseteq V_I^r$.
$u_s \in V_I^r \rightarrow <\overline{u_s,u_b}> \in B^r \rightarrow \exists C_b \in C$,
$<u_s,u_b> \in C_b \rightarrow <u_s,u_b> \in p(C_b,B^r)$
$\rightarrow u_s \in s(\{p(C_s,B^r) \ni C_s \in C\}) = s(r(B^r))$.

(2.3): (2.1), (2.2) $\rightarrow s(r(B^r)) = V_I^r$.

(3): Show $d(r(B^r)) = V_O^r$.
Same as (2) except replace the s set by the d set.

(4): Show $|r(B^r)| \geq 2$.
By Theorem hypothesis.

(5): (1), (2), (3) and (4) $\rightarrow$ $<r(B^r)>$
$\in K[V_I^r \times V_O^r]$. □

Given an arbitrary network it is possible that $|r(B^r)| = 1$ for some B^r; that is, $p(C_s,B^r) = p(C_b,B^r)$, $\forall C_s,C_b \in C$. Then $r(B^r)$ does not constitute a reconfigurable network as defined. To handle this case from an engineering point of view, do the following. If a network contains such a B^r, that part of the network is constant, that is, it has a single state only. So to remove this constant part from the network $K = <C>$ do the following.

(1) Construct separately the constant part $r(B^r)$, $\forall B^r \ni |r(B^r)| = 1$, as a set of nonreconfigurable links.

(2) $K' \triangleq <\{C_m - <v_s,v_b> \mid C_m \in C, \forall <\overline{v_s,v_b}> \in B^r, \forall B^r \ni |r(B^r)| = 1\}>$.

K' then contains only the reconfigurable links. In the following it is assumed that the constant part of the network has been removed already.

If $G[V_I \times V_O] = B^1$, then $K = <r(B^1)>$. In this case, K is a σ-prime network and is not partitionable.

The following Lemmas and Theorems are shown for the case of $G[V_I \times V_O]$ having two components, B^1 and B^2, for reasons of simplicity. They are all applicable to the case of $B^1,B^2,\ldots,B^n$, $n \geq 2$.

Lemma 5.7: Let $\{B^1, B^2\}$ be the set of components of the underlying graph $G[V_I \times V_O]$ of $K \in K[V_I \times V_O]$, $K = <C>$. Let $|r(B^i)| = |C|$, $\forall i$. Then $\exists\ r_\sigma$ such that if
$<C^3> = K(B^1)\ r_\sigma\ K(B^2)$ then $C \subseteq C^3$.

Proof: (1): $|r(B^i)| = |C|, \exists\ i \rightarrow$
This is necessary and sufficient condition for the existence of α.
$p(C_x,B^r) \neq p(C_y,B^r), \forall C_x,C_y \in C, x \neq y, \quad \forall r$.

(2): $<C^3> = K(B^1)\ r_\sigma\ K(B^2) \rightarrow$
$C^3 = \{p(C_s,B^1) \cup p(C_b,B^2) \mid \alpha(p(C_s,B^1)) = p(C_b,B^2), C_s \in C, C_b \in C\}$.

(3): Let
α: $\{p(C_s,B^1) \mid C_s \in C\} \rightarrow \{p(C_b,B^2) \mid C_b \in C\}$,
$\alpha(p(C_s,B^1)) = p(C_s,B^2)$.

(4): $C_s \in C \rightarrow C_s = p(C_s,B^1) \cup p(C_s,B^2)$.

(5): (2), (3) and (4) $\rightarrow C_s \in C^3 \rightarrow C \subseteq C^3$. □

Lemma 5.8: Let $\{B^1, B^2\}$ be the set of components of the underlying graph $G[V_I \times V_O]$ of $K \in K[V_I \times V_O]$, $K = <C>$. Let $|r(B^i)| = |C|$, $\forall i$. Then $\exists\ r_\sigma$ such that if
$<C^3> = K(B^1)\ r_\sigma\ K(B^2)$ then $C^3 \subseteq C$.

Proof: (1): (1), (2), and (3) from Lemma 5.8 *proof*.

(2): $C_x \in C^3 \rightarrow C_x = p(C_x,B^1) \cup p(C_x,B^2)$.

(3): (1) and (2) $\rightarrow C_x \in C^3 \rightarrow C \subseteq C^3$. □

Theorem 5.9: Let $\{B^1, B^2\}$ be the set of components of the underlying graph $G[V_I \times V_O]$ of $K \in K[V_I \times V_O]$, $K = <C>$. Let $|r(B^i)| = |C|$, $\forall i$. Then $\exists\ r_\sigma$ such that $K(B^1)\ r_\sigma\ K(B^2) = K$.

Proof: (1): Let
α: $\{p(C_m,B^1) \mid C_m \in C\} \rightarrow \{p(C_n,B^2) \mid C_n \in C\}$,
$\alpha(p(C_m,B^1)) = p(C_m,B^2)$.
Let $K(B^1)\ r_\sigma\ K(B^2) = <C^3>$.

(2): Lemma 5.7 $\rightarrow C \subseteq C^3$.

(3): Lemma 5.8 $\rightarrow C^3 \subseteq C$.

(4): (2), and (3) $\rightarrow C^3 = C$.

(5): Theorem 3.8 $\rightarrow C^3 = C \rightarrow$
$K(B^1)\ r_\sigma\ K(B^2) = K$. □

Lemma 5.10: Let $\{B^1, B^2\}$ be the set of components of the underlying graph $G[V_I \times V_O]$ of $K \in K[V_I \times V_O]$, $K = <C>$. Let $K(B^1)\ \sigma\ K(B^2) = <C^3>$. Then $C \subseteq C^3$.

Proof: (1): $C_m \in C \rightarrow C_m = p(C_m,B^1) \cup p(C_m,B^2)$.

(2): $<C^3> = K(B^1)\ \sigma\ K(B^2) = <\{p(C_s,B^1) \mid C_s \in C\}>\ \sigma\ <\{p(C_b,B^2) \mid C_b \in C\}> \rightarrow$
$C_m \in C^3 \rightarrow C \subseteq C^3$. □

Theorem 5.11: Let $\{B^1,B^2\}$ be the set of components of the underlying graph $G[V_I \times V_O]$ of $K \in K[V_I \times V_O]$, $K = <C>$. Let $K(B^1)\ \sigma\ K(B^2) = <C^3>$. Let $|r(B^1)| \cdot |r(B^2)| = |C|$. Then $K(B^1)\ \sigma\ K(B^2) = K$.

Proof: (1): By Lemma 5.10 $C \subseteq C^3$.
By Theorem hypothesis $|r(B^1)| \cdot |r(B^2)| = |C^3| = |C| \rightarrow C = C^3$.

(2): By Theorem 3.8 and (1) $\rightarrow$ $K(B^1)\ \sigma\ K(B^2) = K$. □

Theorem 5.12: Let $\{B^1,B^2\}$ be the set of components of the underlying graph $G[V_I \times V_O]$ of $K \in K[V_I \times V_O]$, $K = <C>$. Let $K(B^1)\ \sigma\ K(B^2) = <C^3>$. Let $|r(B^1)| \cdot |r(B^2)| > |C|$. Then $K(B^1)\ \sigma\ K(B^2) \supseteq_c K$ and $K(B^1)\ \sigma\ K(B^2) \neq K$.

Proof (1): By Lemma 5.12 $C \subseteq C^3$.
Theorem hypothesis $|r(B^1)| \cdot |r(B^2)| = |C^3| > |C| \rightarrow C \subset C^3$.

(2): Theorem 3.7 and (1) $\rightarrow$ $K(B^1)\ \sigma\ K(B^2) \supseteq_c K$, and Theorem 3.8 and (1) $\rightarrow$ $K(B^1)\ \sigma\ K(B^2) \neq K$.

□

Note that by definition $K(B^i)$, $\forall i$ is a σ-prime network.

Definition 5.13: If $B^1,B^2,\ldots,B^n$ are the components of $G[V_I \times V_O]$, where $G[V_I \times V_O]$ is the underlying graph of K, then $K(B^1)$, $K(B^2)$,....$K(B^n)$ is a *prime* decomposition of K.

The prime decomposition of K is unique and can be used as a canonical form of the network.

The algorithm is presented below. The input is an arbitrary network $K \in K[V_I \times V_O]$, $K = <C>$, with the constant part removed. The output is one of (1) K is not partitionable, (2) K is τ-partitionable, (3) K is strictly σ-partitionable, (4) K is σ-partitionable. In cases (2), (3), and (4) the algorithm also produces the component networks $K(B^1)$, $K(B^2)$,....$K(B^n)$, in step (6).

Algorithm

Input: $K \in K[V_I \times V_O]$, $K = <C>$

Output: (1): K is not partitionable,
or (2): K is τ-partitionable,
or (3): K is strictly σ-partitionable,
or (4): K is σ-partitionable.

(1) Construct the underlying graph $G[V_I \times V_O]$ of K.

(2) Find components $B^1,B^2,\ldots,B^n$ of $G[V_I \times V_O]$.

(3) If (n = 1) return (1).

(4) Find $p(C_m,B^i)$, $\forall C_m \in C$, $i = 1,2,\ldots,n$.

(5) Find $r(B^i) = \{p(C_m,B^i) \mid \forall C_m \in C\}$, $i = 1,2,\ldots,n$.

(6) Construct $K(B^i) = <r(B^i)>$, $i = 1,2,\ldots,n$.

(7) If $|r(B^r)| = |C|$, $r = 1,2,\ldots,n$)
then return (2).

(8) If $(\prod_{i=1}^{n} |r(B^i)|) = |C|$
then return (3).

(9) Else return (4).

Proof of correctness: The proof is directly implied by Theorems 5.9, 5.11, and 5.12. □

The outputs of the algorithm can be used in the following ways. If the output is "1" (not partitionable), then the system designer will know that the network cannot be divided into individual subnetworks. If the output is "3" (strictly σ-partitionable), then the network can be partitioned and the composition of the component networks will produce a set of correspondences identical to that of the original network. Note that if a network is strictly σ-partitionable it is not τ-partitionable nor σ-partitionable. If the output is "4" (σ-partitionable), then the network can be partitioned and the composition of the component networks will produce a set of correspondences that is a superset of that of the original network. If the output is "2", the network is τ-partitionable. Any network that is τ-partitionable is also σ-partitionable. However, if a network is τ-partitionable then $|r(B^i)| = |r(B^j)| = |C|$, $1 \leq i,j \leq n$, which is not true in general for a σ-partitionable network. Since $|r(B^i)| = |r(B^j)| = |C|$, $1 \leq i, j \leq n$, the number of correspondences in each component network $<r(B^i)>$ is the same ($|C|$) for i, $1 \leq i \leq n$. This property means that the same control decoders can be used in all network components in a τ-partitionable network.

The output of the algorithm applies only to the reconfigurable part of the network because partitionability is defined in terms of a decomposition into "reconfigurable" network components ($|r(B^i)| > 1$). If the original network had some B^i such that $|r(B^i)| = 1$, then those constant component(s) should be added to the network component(s) generated by the algorithm in order to reproduce the original network.

There are less strict definitions of partitionability than the one used here. Future work in this area includes the study of the partitionability of networks if some of the network correspondences are not used, e.g., as can be done with the cube network [13, 14].

VI. Conclusion

In this paper the interconnection network properties of composition, decomposition, and partitionability were analyzed. The partitionability property of interconnection networks for parallel computer systems is important for (1) resource allocation, (2) fault tolerance, and (3) efficient hardware implementation as discussed in the

introduction. The results presented here are valid across all network topologies.

In summary, a general model of interconnection networks was used to describe composition, decomposition, and partitionability properties of networks. An algorithm for network partitioning was presented and proven correct.

VII. References

[1] J. A. Bondy and U. S. R. Murty, *Graph Theory with Applications*, North Holland Publishing, New York, NY, 1976.

[2] J. P. Fishburn and R. A. Finkel, "Quotient networks," *IEEE Trans. Comput.*, Vol. C-31, Apr. 1982, pp. 288-295.

[3] L. R. Goke, "Banyans networks for partitioning multiprocessors systems," Ph.D. Thesis, University of Florida, June 1976.

[4] L. R. Goke and G. J. Lipovski, "Banyan networks for partitioning multiprocessor systems," *Symp. Computer Architecture*, Dec. 1973, pp. 21-28.

[5] C. B. Hanneken, *Introduction to Abstract Algebra*, Dickenson Publishing, 1968.

[6] F. Harary, *Graph Theory*, Addison Wesley Publisher, 1969.

[7] I. N. Herstein, *Topics in Algebra*, Xerox College Publishing, Lexington, MA, 1975.

[8] G. J. Lipovski and M. Malek, "A Theory for Interconnection Networks," Electrical Engineering Department, University of Texas at Austin, TRAC Report 41, Oct. 1982.

[9] M. C. Pease, "An adaptation of the fast Fourier transform for parallel processing," *Journal ACM*, Vol. 15, Apr. 1968, pp. 252-264.

[10] A. P. Reeves and R. R. Seban, "The moment computer," *15th Hawaii Int'l. Conf. System Sciences*, Jan. 1982, pp. 388-396.

[11] R. R. Seban and H. J. Siegel, "Theoretical modeling and analysis of special purpose interconnection networks," *The 4th Int'l. Conf. Distributed Computing Systems*, May 1984, pp. 256-265.

[12] R. R. Seban, H. J. Siegel, and D. G. Meyer, "Data communications in a real-time distributed signal processing system: A case study," *1984 Real-Time Systems Symp.*, Dec. 1984.

[13] H. J. Siegel, "The theory underlying the partitioning of permutation networks," *IEEE Trans. Comput.*, Vol. C-29, Sept. 1980, pp. 791-801.

[14] H. J. Siegel, *Interconnection Networks for Large Scale Parallel Processing: Theory and Case Studies*, Lexington Books, Lexington, MA, 1985.

[15] P. V. Uppaluru, "A theoretical basis for analysis and partitioning of regular SW banyans," Ph.D. Thesis, The University of Texas at Austin, 1981.

[16] C. Wu and T. Feng, *Tutorial: Interconnection Networks for Parallel and Distributed Processing*, IEEE, Computer Society Press, Silver Spring, MD, 1984.

Paper 8

Evaluation of Cube and Data Manipulator Networks

JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING 2, 79–107 (1985)

Evaluation of Cube and Data Manipulator Networks*

ROBERT J. MCMILLEN

Hughes Aircraft Company, Long Beach, California 90810

AND

HOWARD JAY SIEGEL

PASM Parallel Processing Laboratory, School of Electrical Engineering, Purdue University, West Lafayette, Indiana 47907

The interconnection of a large number of processors and other devices to form a parallel/distributed computing system is a research area receiving a great deal of attention. One method is to use a multistage network. This paper compares two classes of multistage networks by examining two representative networks: the Generalized Cube and the Augmented Data Manipulator. The two topologies are compared using a graph model. By interpreting the graphical representations of the networks in different ways, different but functionally equivalent implementations result. The costs of the various implementations are compared taking VLSI considerations into account. Finally, the robustness (fault tolerance) of the different networks is measured and contrasted.

1 INTRODUCTION

The interconnection of a large number of processors and other devices to form a parallel/distributed computing system is a research area receiving a great deal of attention. Many different approaches to the interconnection method have been proposed and discussed including the use of buses [47], hierarchies of buses [44], direct links [13], single-stage networks [21], multistage networks [9, 22, 30, 38], and crossbars [49]. An important aspect of this research is the evaluation and comparison of the proposed approaches [6, 16, 40, 45]. The conclusion most often reached is that the best scheme to use in a particular design depends highly upon the intended application, performance requirements, and cost constraints. Once a connection method

*This work was supported by the United States Army Research Office, Department of the Army, under Grant DAAG29-82-K-0101; the National Science Foundation under Grant ECS 80-16580; and the Air Force Office of Scientific Research, Air Force Systems Commands, USAF, under Grant AFOSR 78-3581. The U.S. Government's right to retain a nonexclusive royalty-free license in and to this paper, for governmental purposes, is acknowledged.

0743-7315/85 $3.00

is chosen (e.g., single-stage network), a specific design must be decided upon and then implemented. During this phase of a system's specification, it is important for the designer to understand fully the differences and similarities between candidate designs.

This work is motivated by an ongoing study of methods to model distributed systems and an examination of networks suitable for use in the PASM [41] and PUMPS [10] systems. Two classes of multistage networks that have been considered for use in these and other systems, cube type and data manipulator type, are investigated in this paper. Specifically, graph models are used to quantify the difference between the Generalized Cube and Augmented Data Manipulator (ADM) networks in terms of cost and robustness (fault tolerance). Graph models are used because they are unencumbered by implementation details and are an excellent tool for representing an essential characteristic of a network: its topology. They also facilitate comparison of this work with other studies (e.g., [5, 19]).

The Generalized Cube and ADM networks are defined in Section II. Their relation to other multistage networks described in the literature is also discussed. Using a graphical representation, the networks' topologies are compared in Section III. In Section IV, two functionally equivalent implementations resulting from two different graph interpretations are examined to compare the cost of each network. Here, using VLSI chips is considered and costs are compared relative to the fraction of a stage that can be implemented on one chip. Finally, Section V contains an analysis of the robustness each network exhibits.

II. The Generalized Cube and ADM Networks

The Generalized Cube network is a multistage cube-type network topology that was introduced as a standard for comparing network topologies [39]. Assume the network has N inputs and N outputs; in Fig. 1, $N = 8$. The Generalized Cube topology has $n = \log_2 N$ stages, where each stage consists of a set of N lines connected to $N/2$ interchange boxes. Each interchange box is a two-input, two-output device. The labels of the input/output lines entering the upper and lower inputs of an interchange box serve as the labels for the upper and lower outputs, respectively. Each interchange box can be set to one of the four legitimate states shown [22].

The connections in this network are based on the *cube interconnection functions* [35]. Let $P = p_{n-1} \cdots p_1 p_0$ be the binary representation of an arbitrary I/O line label. Then the n cube interconnection functions can be defined as

$$cube_i(p_{n-1} \cdots p_1 p_0) = p_{n-1} \cdots p_{i+1} \bar{p}_i p_{i-1} \cdots p_1 p_0,$$

where $0 \le i < n$, $0 \le P < N$, and $\bar{p}_i$ denotes the complement of p_i. This

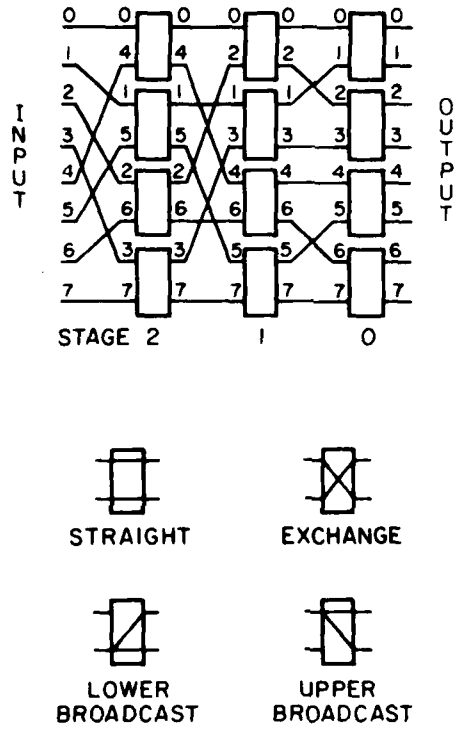


FIG. 1. Generalized Cube network for $N = 8$ [37]. The four legitimate states of an interchange box are shown.

means that the $cube_i$ interconnection function connects P to $cube_i(P)$, where $cube_i(P)$ is the I/O line whose label differs from P in just the ith bit position. Stage i of the Generalized Cube topology contains the cube interconnection function. That is, it pairs I/O lines that differ in the ith bit position.

The ADM network is shown in Fig. 2 for $N = 8$. It is based on Feng's data manipulator [15]. In this network, a *stage* consists of N *switching elements* or *nodes* and the $3N$ data paths that are connected to the inputs of a succeeding stage. Each node can connect one of its inputs to one or more of its outputs. At stage i of the ADM network, $0 \leq i < n$, the first output of node j is connected to the input of node $(j - 2^i) \bmod N$ of the next stage; the second output is connected to the input of node j; and the third output is connected to the input of node $(j + 2^i) \bmod N$. Because $(j - 2^{n-1})$ equals $(j + 2^{n-1}) \bmod N$, there are actually only two distinct data paths instead of three from each node in stage $n - 1$ (in the figure, stage 2). There is an additional set of N nodes at the output stage.

Both of these networks are based on the *PM2I interconnection functions* [35]. There are $2n$ of these functions defined by $PM2_{+i}(j) = j + 2^i \bmod N$

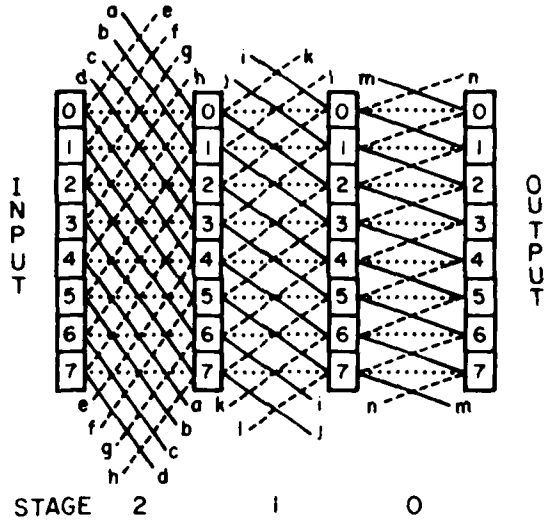


FIG. 2. Augmented Data Manipulator network for $N = 8$ [37]. (Lowercase letters represent end-around connections.)

and $PM2_{+i}(j) = j - 2^i \bmod N$ for $0 \le j < N$, $0 \le i < n$, where $-x \bmod N = N - x \bmod N$. (Note $PM2_{+(n-1)} = PM2_{-(n-1)}$.)

A number of systems have been proposed and/or built that use multistage networks (e.g., [7, 8, 24, 34, 41]). Among the networks that have been proposed are the ADM [38], baseline [48], binary n-cube [30], data manipulator [15], Gamma [29], Generalized Cube [39], inverse ADM (IADM) [27], omega [22], STARAN flip [9], and SW-banyan [19]. Studies have shown that the baseline, binary n-cube, Generalized Cube, omega, STARAN flip, and SW-banyan ($S = F = 2$) networks are all topologically equivalent [31, 36, 37, 42, 48]. Differences between these networks are due to proposed control schemes, whether or not a broadcast capability is included, and the method used to number input and output ports. All of these networks belong to the general class of cube-type networks. Because of the similarities among these networks, a designer is not faced with choosing between six different networks; rather the choice is whether or not to use a cube-type network.

The data manipulator, ADM, IADM, and Gamma networks are topologically identical. The differences between these networks are the control scheme, order in which stages are traversed, and switch complexity. The switches in each stage of the data manipulator are divided into two groups. Each group receives an independent set of control signals and all switches in a group respond identically. Each switching element of the ADM, IADM, and Gamma networks is controlled individually. The stages of the IADM and Gamma networks are traversed in an order opposite to that of the ADM and data manipulator. Also, the Gamma network's switching elements are 3×3

crossbars (as opposed to selecting one input at a time). One property that these networks have is that for all nontrivial source/destination pairs (i.e., source address ≠ destination address) there are multiple paths through the network. For that reason, none of the networks is a member of the general banyan class [19].

The capabilities of the Gamma network are a superset of the ADM and IADM networks. It has been shown in turn that their capabilities are a superset of all the cube-type networks as well as the data manipulator network [36, 37, 42]. Data manipulator-type networks, however, are more complex than cube-type networks.

A common feature of all cube-type networks is that there is exactly one path through the network for each source/destination pair. This property makes control schemes simple but any single failure of a link or switch will disallow the use of any path requiring the failed component.

Thus there exists the classic trade-off between cost and performance when choosing between the two network types. In this paper, the network types are compared, using one representative network from each type: the Generalized Cube and the ADM. Both networks have the same number of input and output ports and individual switching element control. Routing tag schemes are available for the networks [22, 28, 38, 39], so it is assumed that they are used to implement network control.

Some aspects of the Generalized Cube and the ADM networks have been compared elsewhere. The ability of the ADM network to perform all the functions a Generalized Cube can was demonstrated in [42]. In [1], the total number of unique permutation connections each network can perform was compared. In [5], graph models were used to study multistage interconnection networks which have the "buddy property" (cube-type networks have that property) and other networks including the ADM. In that paper emphasis was on comparing the networks' permutation capabilities. This paper is concerned with comparing cost and robustness or inherent fault tolerance. Cost is examined from two points of view. The first is the common method of counting links and switching nodes. In this case, the graph model with a consistent interpretation (two are possible) is used to ensure a "fair" comparison. The second point of view is oriented toward VLSI considerations. Modules for each network requiring roughly the same number of pins are compared. The change in relative cost is also examined when as much as one whole stage is placed on one chip. Robustness is measured by calculating the average number of network inputs and outputs affected by the removal of a single link or switching element. The calculations are performed for both of the graph interpretations to be defined.

III. Graph Modeling: A Common Basis for Comparing Networks

Graph models have been used by Goke and Lipovski [19] as the basis for defining a class of networks called banyans. The graphs used to represent

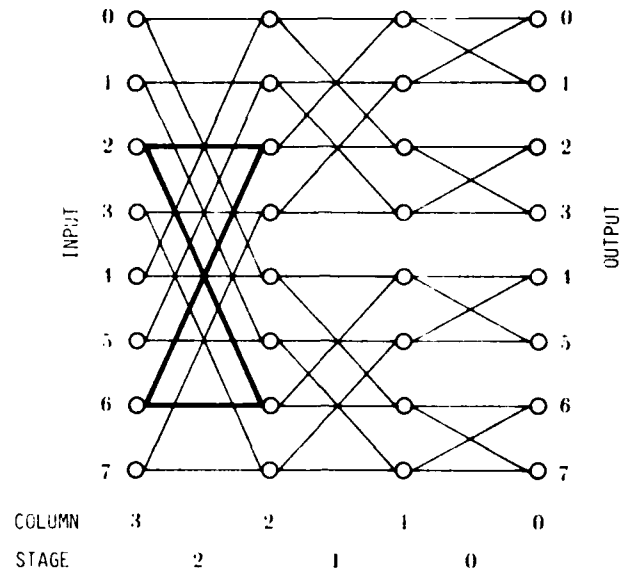


FIG. 3. Graphical representation of the Generalized Cube network for $N = 8$.

these networks consist of *nodes* connected by *directed arcs*. By definition, in a banyan there is one and only one path from input to output [19]. In this paper the arcs are undirected and there is no restriction on the number of paths from input to output.

It has been observed [20, 23] that the Generalized Cube network (Fig. 1) has the graphical representation shown in Fig. 3. This graph also represents an SW–banyan (with $S = F = 2$). The graph can be interpreted a number of different ways. One is to treat each node (vertex) (a circle in the figure) as a switch and each arc (edge) (a line in the figure) as a link. To model the network's behavior under this interpretation, the switch (node) shown in Fig. 4a should only connect one of the input links, a *or* b, to one of the output links, c *or* d. An implementation based on this interpretation, for an N input/output network, would consist of $n + 1$ stages of N switches, with $2N$ lines between stages. The TRAC reconfigurable, multimicroprocessor system contains an SW–banyan constructed from switches of this type (but that have two incoming and three outgoing links, i.e., $S = 2$ and $F = 3$) [32].

A second interpretation of the graph in Fig. 3 is to treat the nodes as links and the arcs as forming interchange boxes. For example, the thickened lines in Fig. 3 can be considered to represent the interchange box with inputs 2 and 6 (compare this to Fig. 1). In this case the SW–banyan implementation would have the same structure as specified here for the Generalized Cube (assuming a bidirectional network). This interpretation is illustrated in Figs. 4b and c.

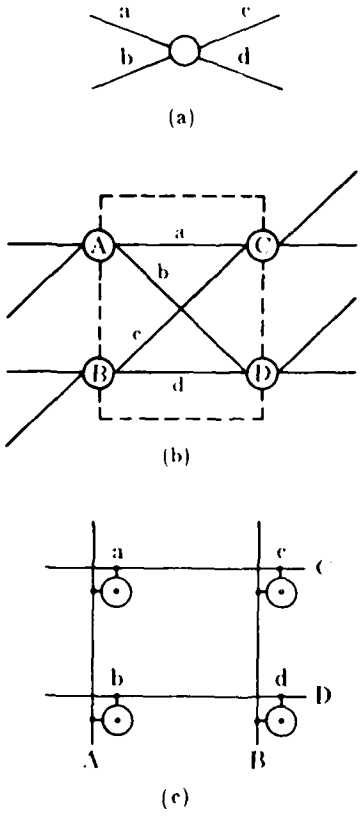


FIG. 4. (a) A node from the graph representing the Generalized Cube network. When equated with a switch, input *a or b* can be connected to output *c or d*. (b) Four nodes from the graph. When the arcs, *a*, *b*, *c*, and *d* are equated with switches, a 2 × 2 crossbar is obtained. (c) The components of a crossbar that correspond to the graph in (b).

Each of the arcs labeled a through d in Fig. 4b acts as a crosspoint switch in Fig. 4c. When viewed this way, the portion of the graph within the dashed lines of Fig. 4b behaves as a 2 × 2 crossbar or interchange box. If a and d are "on," the straight setting is obtained; b and c "on" corresponds to exchange; a and b "on" corresponds to upper broadcast; and c and d "on" corresponds to lower broadcast. Conflict occurs if a and c or b and d are on at the same time. It will be shown in the next section that implementations based on the first and second graph interpretations are functionally equivalent.

A third possible interpretation of the graph in Fig. 3 is to equate nodes with 2 × 2 interchange boxes and arcs with links. In that case, Fig. 3 would represent a size $N = 16$ Generalized Cube network. This interpretation will not be discussed further in this paper.

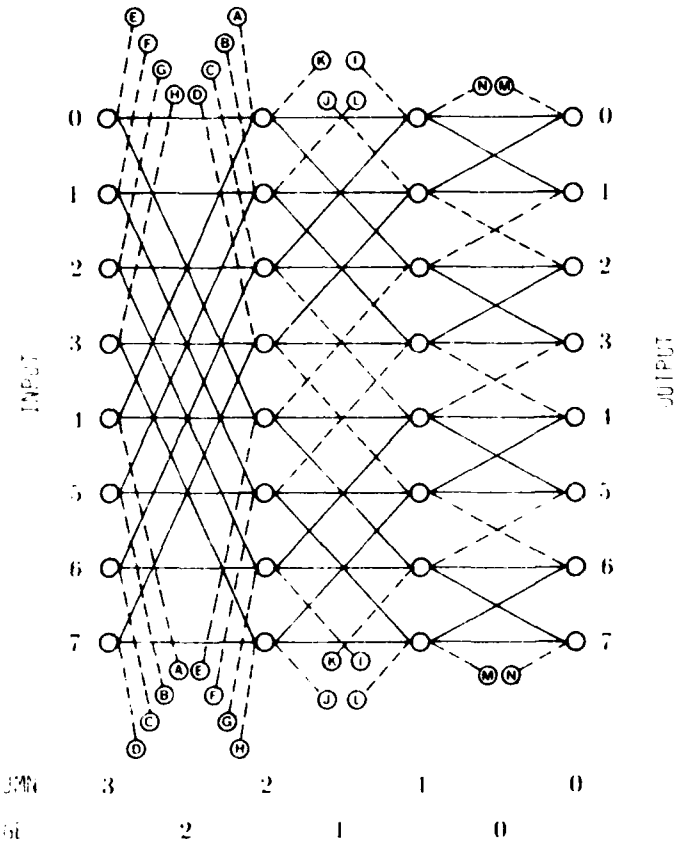


FIG. 5. Graphical representation of the Augmented Data Manipulator for $N = 8$.

The graphical representation of the ADM network (Fig. 2) is shown in Fig. 5. Since there are multiple paths from input to output, this is not a banyan graph. This graph can be obtained by adding the dashed lines shown in Fig. 5 to the graph in Fig. 3.[1] When switches are equated with nodes, the network depicted in Fig. 2 is obtained. When switches are equated with arcs, the network looks like that shown in Fig. 6. In the figure, two nodes directly connected by a solid line between stages are represented by a single node in Fig. 5. Note that the labels on end-around connections in both Fig. 5 and Fig. 6 are attached to the same arcs (links) in the network. This second type of

[1]We first published this observation in October 1982, in the Proceedings of the Third International Conference on Distributed Computing Systems, in a preliminary version of this material. It was also discovered independently and published in [5].

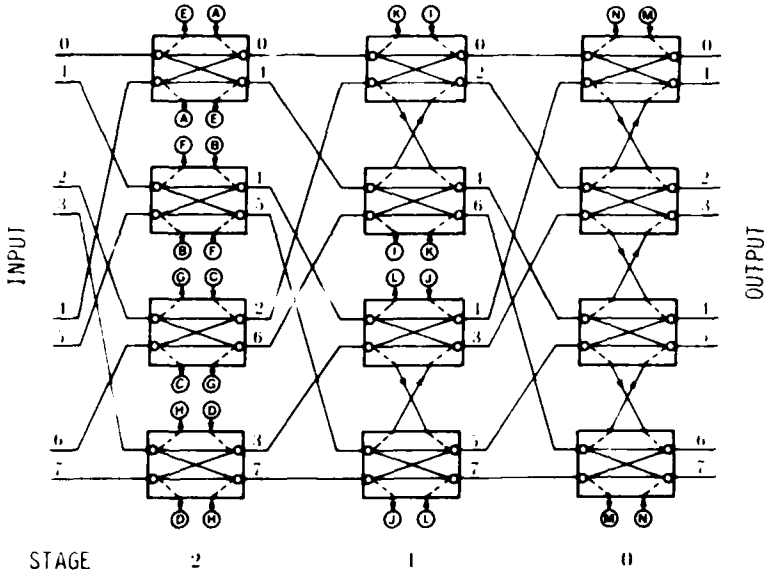


FIG. 6. Implementation of the Augmented Data Manipulator for $N = 8$ when the graph of Fig. 5 is interpreted with arcs equated to switches.

implementation is examined in [43], where LSI packaging of network building blocks is discussed.

Though the same ADM network is represented, Figs. 2 and 6 look rather different. Depending upon which representation is chosen, a comparison with the Generalized Cube in Fig. 1 could produce different conclusions. Comparing Figs. 1 and 2, one might conclude that, in addition to having an extra column of switches, the ADM has twice as many switching nodes and three times as many links as the Generalized Cube network. It would be easy to decide that the ADM network is considerably more expensive. On the other hand, comparing Figs. 1 and 6, it appears the only difference is N extra links that interconnect switches within each stage of the ADM network. The latter comparison is more accurate because the network depictions of Figs. 1 and 6 are based on the same interpretation of the networks' respective graphs. Thus when making comparisons, it is important to compare either graphical representations or consistent interpretations of those graphs. In the next section, the latter is done for both interpretations, so that the resulting implementations can be compared as well.

IV. Cost Comparison

A. *Introduction*

The purpose of this section is to compare the cost of the Generalized Cube network to that of the ADM network. To do this, implementations of each

network are examined. Two different criteria are used in the comparison. First, hardware requirements are examined. Since two basic implementations are possible for each network, to be fair, only implementations corresponding to the same graph interpretation are compared. Then, since VLSI implementation is being considered, the total number of data pins available on a chip is held constant and chip counts are compared for all the different implementations. It would be desirable to compare the gate densities required for each chip; however, that requires having a detailed design for each. In lieu of such details, the attempt was made to compare chips with comparable major architectural features (e.g., queues) which presumably require the same amount of logic and which can be compared at a gross level.

Although the discussion presented here is in terms of integrated circuit chips, it is not restricted to any particular technology. It is only presumed that a network is constructed from modular elements with I/O facilities (ports) proportional to that portion of the network graph (with an appropriate interpretation) intersected by the boundary of the module. For example, in the future, an I/O port may consist of a laser diode and a single optical fiber instead of many parallel wires.

B. *Hardware Realizations*

There are two basic ways to implement multistage networks. They can be circuit switched or packet switched. In circuit switching, a complete path is established from input to output and must be held for the duration of the communication. Circuit switching is often used when processors are connected to the network inputs and memories are connected to the outputs. Designs for circuit-switched interchange boxes have been discussed in [11, 26, 43]. In packet switching, messages are decomposed into packets which each make their way from stage to stage until the output is reached. This method is often used in configurations that connect processing element (processor/memory pair) j to input j and output j of a unidirectional network. Packet-switched network switching element designs have been discussed in [14, 26, 46].

In the remainder of this paper, implementations will be discussed primarily in terms of packet switching. Circuit-switched versions can be obtained by replacing any queues shown with buses. Other than this, remaining differences are in the control logic; however, the logic is shown only at the block diagram level. Only key elements of the implementations to be discussed are included since many variations of the basic designs are possible. For more detail see [14, 26, 46].

C. *Generalized Cube*

Figure 7 shows two designs for a Generalized Cube switching element. Figure 7a results when switches are equated with nodes in the graph (this corresponds to Figs. 4a and 3). One of the two inputs is selected depending

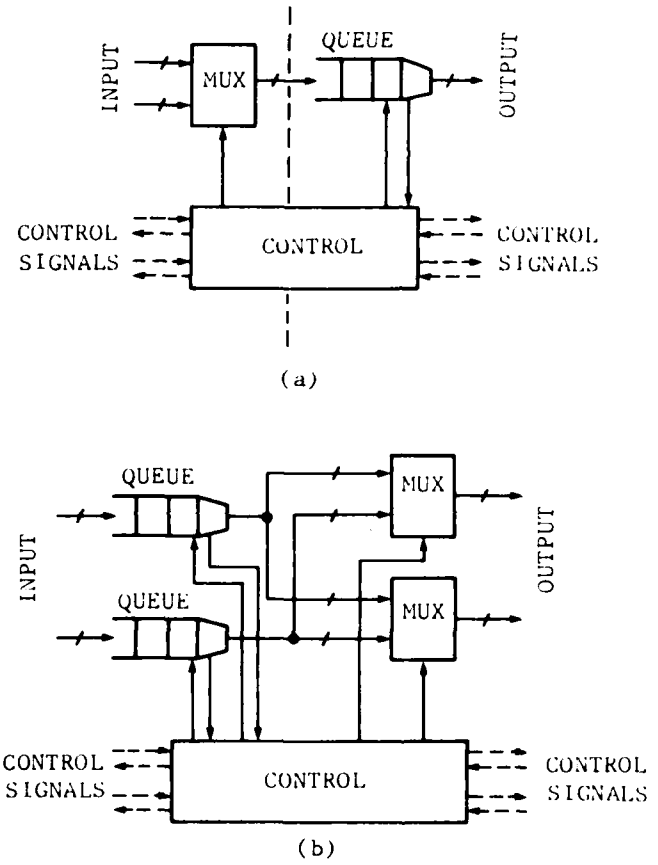


FIG. 7. Implementation of Generalized Cube switches. (a) Node = switch interpretation. (b) Arc = switch interpretation.

on the requests (if any) received by the (left half of the) control logic, which handles any needed arbitration. A single output link is shown, but it is to be connected to *two* other switches as shown in Fig. 8. A bit in the routing tag is examined by the control logic, which then determines to which switch a request for access should be made. The (right half of the) control logic maintains the queue, interprets the routing tag, generates access requests, and receives grants for access requests. Switches that implement nodes in column 3 of Fig. 3 only contain hardware to the right of the dashed line in Fig. 7a. Switches that implement column 0 nodes only contain hardware to the left of the dashed line. A detailed design of this type is discussed in [32].

If arcs in the graph are equated with switches, then four arcs form a 2 × 2 crossbar or interchange box (see Figs. 4b and c and 1). An implementation

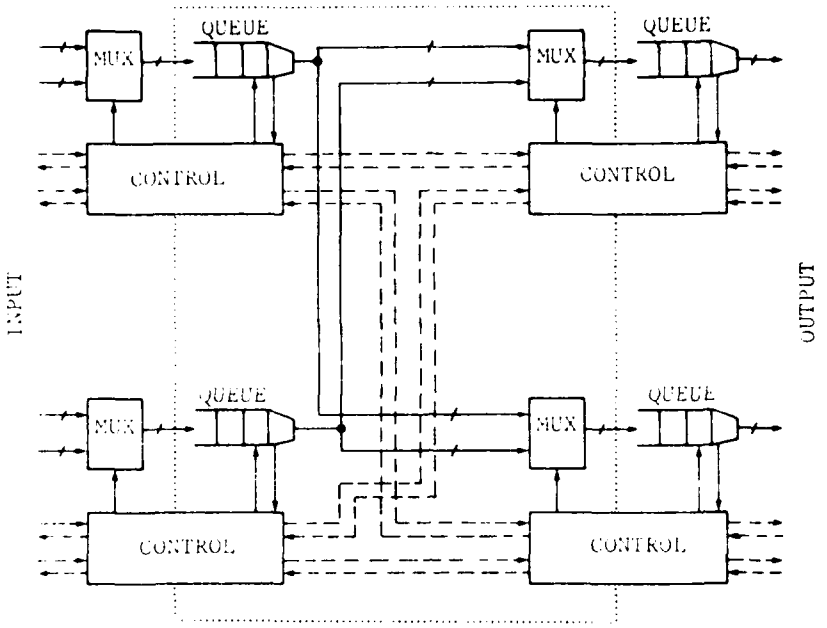


FIG. 8. Four switches from Fig. 7a combined to form one switch (within dashed lines) equivalent to that in Fig. 7b.

for this is shown in Fig. 7b. Here two input queues are required. As long as a given queue is not full, incoming packets for that queue will be accepted. Logic is required to handshake with other interchange boxes, maintain two queues, and interpret the routing tags at the head of each queue. This logic only interprets the tags in order to request the desired settings for the multiplexers. Logic asociated with the multiplexers performs any necessary arbitration. It also makes appropriate requests of other interchange boxes once the multiplexers are set. Different protocols and design variations for this type of switching element are discussed in [26]. The performance of networks implemented with these interchange boxes has been studied in [3, 14, 25].

The equivalence of two networks implemented with the two kinds of switching nodes is illustrated in Fig. 8. Four of the switching elements shown in Fig. 7a are connected as prescribed by the graph in Fig. 3. It can be seen that the hardware within the dashed lines is identical to that shown for the interchange box in Fig. 7b. The handshaking lines (directed dashed lines) shown connecting control units are equivalent to internal connections between the tag interpretation and queue control logic and the arbitration and output request logic in the control unit of Fig. 7b. It is thus apparent that the same total amount of hardware is required for either implementation, but that

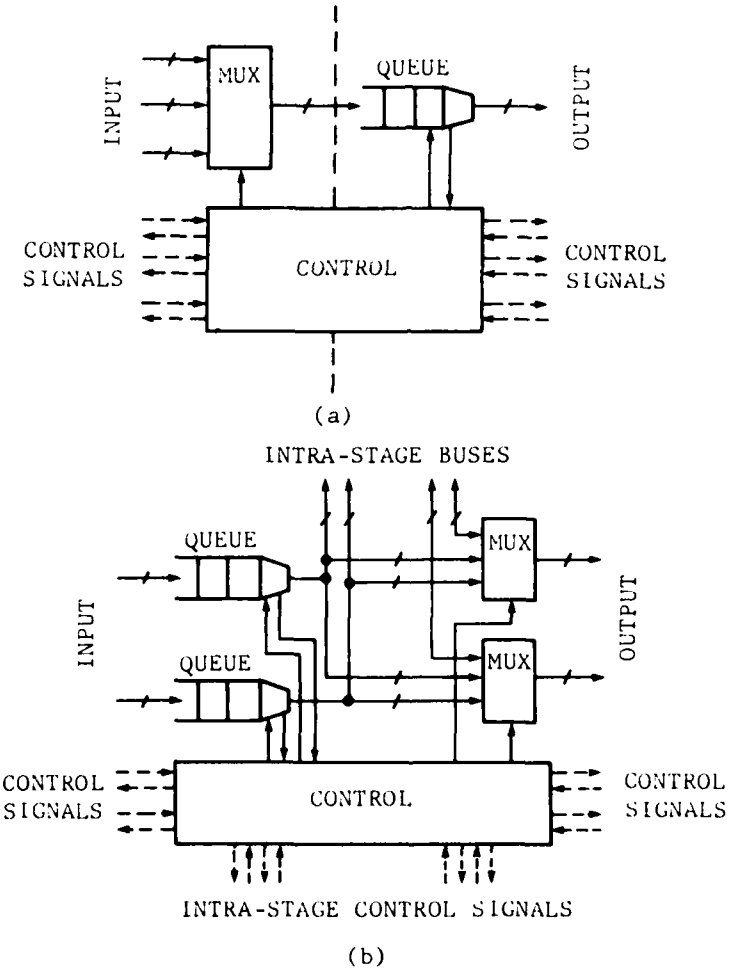


FIG. 9. Implementation of Augmented Data Manipulator switches. (a) Node = switch interpretation. (b) Arc = switch interpretation.

the two graph interpretations lead to different network building blocks or packagings for the components.

D. *Augmented Data Manipulator*

Two implementations for the ADM network are shown in Fig. 9. Figure 9a results from equating the nodes of Fig. 5 with switches. In this design, the multiplexer selects from among three input links and the output link is connected to three other switches. The control signals shown on the output side

in Fig. 9a are used to determine which of the switches is to read the data from the output link. A broadcast is performed by selecting more than one switch. The basic routing tag scheme for the ADM network [28] requires the routing tag logic to examine two bits, so it is slightly more complex than that required in the Generalized Cube. As with the Generalized Cube, the switches implementing nodes in columns 0 and 3 of Fig. 5 only require the logic to the left and right, respectively, of the dashed line in Fig. 9a. This was also observed in [15].

If arcs are equated with switches, an implementation similar to the interchange box is obtained as shown in Fig. 9b. Here, however, the outputs from the queues must be connected to multiplexers in two other switching elements (as shown in Fig. 6) via *intrastage buses*. Similarly, the two multiplexers shown here must accept connections from the queues of two other switching elements. Two control signals must also accompany each of the intrastage buses.

E. *Comparison*

An approximate cost comparison between the Generalized Cube and the ADM network can be made by comparing their respective switching elements. Since the choice is arbitrary, Figs. 7a and 9a will be compared. Both require a single queue. If the cost of the queue and its associated control logic dominates the cost of the switching element, then the ADM switch will cost only slightly more than a Generalized Cube switch in a discrete implementation. On the other hand, for a circuit-switched implementation, the multiplexer and control logic in an ADM switching element will cost about 50% more than that required in a Generalized Cube switching element.

The perspective changes somewhat when implementing these four designs in VLSI is considered. Input/Output (I/O) requirements and logic/pin ratio become important considerations. For constructing a Generalized Cube network, the interchange box in Fig. 7b is a better choice than the switch in Fig. 7a. The interchange box (Fig. 7b) has approximately 33% more pins but approximately 100% more logic than the switch (Fig. 7a). For the ADM network, the logic/pin ratio is nearly the same for both of the designs in Fig. 9. The design in Fig. 9b has approximately twice as many pins and twice as much logic as that shown in Fig. 9a. The extra links that give the ADM network its superior capabilities over the Generalized Cube require a larger number of pins on the VLSI chips being considered.

The design of Fig. 7b and that of 9a have approximately the same number of pins. If this number of pins (due to the data path width) is near technological limits (and thus the design of Fig. 9b will not fit on one chip), then the Generalized Cube interchange box is superior due to the logic/pin ratio. Assuming the cost of two chips with the same number of pins is about the same, an ADM network would be more than twice as expensive as a Generalized Cube network of the same size (when realized with these two re-

spective chips). The logic/pin ratio of the ADM chip (Fig. 9a) can be improved considerably by implementing extra capabilities the ADM network is known to support [27, 28]. These capabilities include dynamic rerouting of blocked messages and stage look-ahead with rerouting for blockage prediction. None of the additional features requires any extra pins. The additional capabilities are possible because of the extra paths between input and output and thus are *not* available for the Generalized Cube network.

The cost difference between the two implementations of each network due to pin limitations can be further quantified. Assume that one switching element is implemented on one chip and that the chips are bit sliced. For the sake of modularity, in the node-equals-switch implementation of both networks, this means the chip will be more complex than necessary for the switching elements in the input and output columns (3 and 0 in Figs. 3 and 5).

Let D be the number of pins available on the chip for data path connections and P be the number of I/O ports required by the switching element (see Figs. 7 and 9). The D/P is the number of pins available per port. It is assumed that data pins dominate the total pin count and that the chip has the capacity to accommodate the small number of control and power pins also needed. If the network data path width is W, the $W \cdot P/D$ is the number of chips required to construct one switching element. Multiplying this by the number of switching elements needed to implement the network gives the total chip count. The expressions for the chip count for the four implementations as a function of W, D, N, and n are given in Table I. A crossbar is included for comparison. The arc-equals-switch (interchange box) implementation of the Generalized Cube gives the lowest count regardless of the values of W, D, and N. As an example of the number of chips required in networks of size $N = 16$ and $N = 64$, assume the network path width is $W = 32$ bits and there are a total of $D = 64$ pins available on the chip for data connections. The resulting counts are shown in Table I.

TABLE I
COMPARISON OF CHIP COUNTS FOR TWO IMPLEMENTATIONS OF GENERALIZED CUBE AND ADM NETWORKS[a]

Network	Implementation	P	Chip count	$W = 32, D = 64$	
				$N = 16$	$N = 64$
Generalized Cube	Node = Switch	3	$3(W/D)N(n+1)$	120	672
	Arc = Switch	4	$2(W/D)Nn$	64	384
ADM	Node = Switch	4	$4(W/D)N(n+1)$	160	896
	Arc = Switch	8	$4(W/D)Nn$	128	768
Crossbar	Crosspoint linking two buses	2	$2(W/D)N^2$	256	4096

[a] P is the number of I/O ports per switching element, W is the network path width, D is the number of data pins per chip, and $n = \log_2 N$.

As advances in packaging technology continue, the cost difference between the Generalized Cube and the ADM will narrow considerably until the ADM is more cost effective. To see this, examine Fig. 6. The larger the number of switching elements (of the type in Fig. 9b), in the same stage, that can be placed on a single chip, the more intrastage buses can be internalized. This reduces the I/O overhead of the extra links. If a whole stage can be placed on one chip, then the ADM network requires the same number of chips and connections between chips as the Generalized Cube network. The assumption here is that the chip circuit density is not sufficient to support a crossbar but it will accommodate more logic than one stage of a Generalized Cube requires. The ADM network's structure thus fills a gap between the cube-type networks and crossbars. Until very large portions of an interconnection network can be placed on a single chip, it is clear that the ADM network will be more expensive to implement than the Generalized Cube, though the difference will continue to decline. Thus, it is important to determine the networks' cost effectiveness. It has already been pointed out that the ADM's capabilities are a superset of the Generalized Cube's. Another factor that is becoming more important as the construction of enormous systems is considered will be discussed in the next section: robustness of inherent fault tolerance.

V. Robustness: A Comparison of Degradation under Component Failure

A. Introduction

In this section, the robustness of each network is measured by removing a single component (link or switch) and counting the number of input and output ports that are affected. An input port is considered affected if it cannot send a message to all output ports. An output port is considered affected if there is at least one input port from which it cannot receive messages. Since the number of ports affected varies with the location of the removed component, averages are computed. Calculating the averages for the Generalized Cube network is relatively straightforward. Calculating the averages for the ADM network is complicated considerably by the varying numbers of multiple paths between ports of the network. Extensive use of and extension to the theoretical result in [28] were required to obtain the closed-form solutions presented here. To streamline this presentation, however, most of the mathematical derivations appear in the Appendix.

The average number of affected ports is calculated for both implementations of each network. These calculations are performed using two different rules for counting affected ports. The first rule requires all I/O ports to be considered. Under this rule, it has been shown that some permutation

connections can be routed around a faulty link in the ADM network, but this is not true in general [38].

The second rule allows "severely" affected ports to be disabled and thus not included in the count. It is implemented as follows. Referring to the graphs in Figs. 3 and 5, if a straight (or horizontal) arc at level j is removed, then input port j and output port j are disabled. If links are equated with arcs, one pair of I/O ports is disabled. If switches are equated with arcs, since two straight arcs are included in each switching element (Figs. 7b and 9b), two pairs of I/O ports are disabled. Thus, in Figs. 1 and 6, the I/O ports whose addresses correspond to the output labels on a given switching element are disabled if that switching element fails.

This second rule takes into account a practical system response to a network fault: the disabling of some components so that operation can continue, but in a degraded mode. This is feasible if the network is used for asynchronous communication by cooperating processors (MIMD mode [18]). If the network is used in a synchronous mode to establish permutation connections (SIMD mode [18]) disabling some components is not feasible. However, if the system is partitionable so that subsets of the processors, called submachines, operate synchronously but independent of other submachines, then certain submachines can be disabled when a fault occurs. PASM [41] and TRAC [34] are systems with this capability.

Since robustness is useless unless it can be exploited, it is implicit that faults can be detected and diagnosed and that the system can continue to function once a fault is detected. Detection and diagnosis have been investigated in [4, 17, 33, 39]. The latter requirement implies that measuring robustness is only meaningful for MIMD and partitionable SIMD environments.

The results using the first rule are shown in Table II and those using the second rule in Table III. Two examples of how to calculate the expressions

TABLE II
AVERAGE NUMBER OF AFFECTED I/O PORTS IN THE GENERALIZED CUBE AND ADM NETWORKS WHEN LINKS AND SWITCHES ARE REMOVED[a]

	Node = Switch		Arc = Switch	
Failure	Link	Switch	Link	Int. box
Generalized Cube	$\frac{2N-2}{n}$	$\frac{4N-2}{n+1}$	$\frac{4N-2}{n+1}$	$\frac{4N-4}{n}$
ADM	$\frac{N+n-1}{3n}$	$\frac{2N+n}{n+1}$	$\frac{2N+n}{n+1}$	$\frac{7N-8}{2n}+1$
Cube/ADM	≈ 6	~ 2	~ 2	1.14

[a] All ports are considered. Node = switch implementation corresponds to Figs. 3 and 2. Arc = switch implementation corresponds to Figs. 1 and 6.

TABLE III
AVERAGE NUMBER OF AFFECTED I/O PORTS IN THE GENERALIZED CUBE AND ADM NETWORKS WHEN LINKS AND SWITCHES ARE REMOVED[a]

	Node = Switch		Arc = Switch	
Failure	Link	Switch	Link	Int. box
Generalized Cube	$\frac{2N}{n} - 1$	$\frac{2N}{n+1} - 2$	$\frac{2N}{n+1} - 2$	$\frac{2N}{n} - 4$
ADM	0	0	0	$\frac{3N}{2n} - 3$
Cube/ADM	∞	∞	∞	≈ 1.33

[a] Severely affected ports are disabled and not counted.

in Table II are shown in the next subsection. The remaining derivations for both tables are presented in the Appendix. It should be noted that the entries in both tables under the "Node = Switch" column for a switch failure and under the "Arc = Switch" column for a link failure are identical. This is because both situations correspond to removing a single node from the graphical representation. Removing a link from the node-equals-switch implementation corresponds to removing a single arc from the graph, whereas removing an interchange box from the arc-equals-switch implementation corresponds to removing four or eight arcs from the graph, considerably different situations.

B *Fault Effect Analysis Counting All Affected Ports*

Here the effects of a link fault are analyzed in detail for the Generalized Cube network and then for the ADM network. For the former, assume there is a link failure in stage i, the first rule applies, and the network is implemented by equating nodes with switches (Fig. 3). To see which inputs are affected, start with the failed link and move backward toward the input, tracing all links that are connected to the failed one. The number of affected inputs corresponds to the number of traced links in stage $n - 1$. In general this number is 2^{n-i-1}. For example, if the link at level 4, stage 1 (Fig. 3), fails, inputs 0 and 4 are affected. The number of affected outputs is calculated by tracing links from the failed one to those to which it is connected in the last stage, i.e., stage 0. This number is expressed as 2^i. For example, failure of the link at level 6, stage 2 (Fig. 3), affects outputs 4, 5, 6, and 7. To calculate the average number of affected I/O ports, given a single link failure, a sum of these two terms taken over all stages is computed:

$$\frac{1}{n}\sum_{i=0}^{n-1}(2^{n-i-1}+2^i) = \frac{2}{n}\sum_{i=0}^{n-1}2^i = \frac{2(N-1)}{n}.$$

As a second example, consider the case of a link failure in the ADM network, using the first rule, and implemented by equating nodes with switches (see Fig. 2). A property of the ADM network is that there are at least two paths between every nontrivial (input address $\neq$ output address) input/output pair [28]. One of the existing paths consists of plus and straight links only and is called *positive dominant*. There is another path that consists of minus and straight links only and it is called *negative dominant*. The portions of the positive and negative dominant paths that are distinct depend on the relationship between the addresses. If they agree in the low-order $i + 1$ bits, then the paths converge at the input to stage i and follow the same set of straight links in stages i through 0. (The paths will be distinct in stages $n - 1$ through $i + 1$.) Thus if a nonstraight link fails, none of the I/O ports are affected because there will be a distinct path of the opposite dominance that avoids that link. (Routing schemes have been proposed that allow messages to *dynamically* switch between positive and negative dominant paths as they traverse the network [27, 28], allowing them to avoid busy or faulty links and switches.) If a straight link in stage i at level j fails, then all the input ports whose low-order $i + 1$ bits agree with output port j's low-order $i + 1$ bits will not be able to send a message to output j. There are 2^{n-i-1} such input ports. The other input ports can communicate with output port j since their paths to j do not converge until reaching a stage less than i. No output port other than j is affected by the failure. To see this, consider output port $k \neq j$. All input ports must be able to communicate with k. They can be divided into two classes: (1) those whose addresses agree with k's in less than $i + 1$ low-order bits; and (2) those whose addresses agree with k's in at least $i + 1$ low-order bits. In the first case, either a given path from the input to output k does not include the faulty straight link (in stage i, level j) or if it does, there is another path of opposite dominance that does not. In the second case, in stages i through 0 the required path uses straight links; however, they are all at level k. Thus all inputs can communicate with output k so k is unaffected. As an example, suppose the straight link in stage 1, level 4 (in Fig. 2), is bad ($i = 1, j = 4$). Consider three different situations: communication from inputs 0 and 4 to output 4, from input 0 to output 5, and from input 1 to output 5. Output 4 will be unable to receive messages from inputs 0 and 4, since 0 and 4 agree with j in the low-order $i + 1$ bits (2 bits). All the other output ports are unaffected. Consider connecting input 0 to output 5. Even though the positive dominant path, $+2^2$, straight, $+2^1$, from input 0 to output 5 includes the bad straight link, a message can simply take the negative dominant path, straight, -2^1, -2^0. Input 1 agrees with output 5 in the two low-order bits (bits 0 and 1) and therefore requires straight links in stages 0 and 1. However, the required links are at level 5, and thus the faulty straight link is not required.

The average number of I/O ports affected by a bad *straight* link, under the first rule, is calculated by adding the number of affected input and output ports

(as a function of the stage in which the fault is located) and summing over all stages:

$$\frac{1}{n}\sum_{i=0}^{n-1}(2^{n-i-1}+1)=\frac{1}{n}\sum_{i=0}^{n-1}(2^{i}+1)=\frac{N+n-1}{n}.$$

Since the failure of a $+2^i$ or a -2^i link does not affect any I/O ports, if link failures are equally likely, then the average over all links is one-third of the above value.

In Table II, the ratio of the average number of affected I/O ports in the Generalized Cube to those in the ADM is computed. Regardless of network size, in the node-equals-switch implementation, a link failure in the Generalized Cube network affects six times as many ports, on the average, as a link failure in the ADM. A given switch failure affects twice as many ports. In the arc-equals-switch implementation, a link failure in the Generalized Cube network affects twice as many ports as the same failure in the ADM network. An interchange box failure affects 1.14 times as many ports.

C. *Discussion of Fault Effects with Some Disabled Ports*

The measurement using the first rule is a very conservative indication of the robustness of the ADM network. Table III shows that under the second rule, the ADM network is very robust. When the pair of I/O ports connected to the network at the level of the failure is disabled in the node-equals-switch implementation, none of the remaining ports is affected by a link or a switch failure. A failure can only eliminate one of at least two paths that are always available between the enabled I/O ports (as illustrated in the example above connecting input 0 to output 5). In the arc-equals-switch implementation, link failures have no effect on enabled I/O ports, because (as pointed out earlier) the situation is equivalent to removing a switch in the node-equals-switch implementation. However, "interchange box" failures do affect some enabled ports. The reason this is the case is that there are situations in which both paths between input and output ports pass through the same box. It so happens that these situations only occur in networks larger than size $N = 8$. The full details are presented in the Appendix.

The entries in Table III for the Generalized Cube network are calculated in a fashion similar to those in Table II. The derivation for each entry is given in the Appendix.

The "interchange box" implementation fault analysis for the Generalized Cube and ADM networks considers only the worst case; i.e., the entire box is faulty. Whether both paths are acutally blocked due to a single fault in a real implementation depends on the nature of the fault. Assume that one interchange box is implemented on a single integrated circuit chip. If two data lines internal to the chip and coming from the same input become shorted, this will have no effect on the other internal data path and it is not necessary to

assume that the whole interchange box has failed. On the other hand, a mechanical failure could affect enough of the chip to render the entire device unusable. From a reliability point of view, this analysis shows that the implementation in Fig. 9a (which corresponds to the network in Fig. 2) is to be preferred over that in Fig. 9b (which corresponds to the network in Fig. 6). Since the logic/pin ratio is roughly the same for both implementations, nothing is lost. However, the total component count will be higher, leading to a less physically compact implementation.

The robustness measures for the ADM network are equally applicable to all the data manipulator-type networks with individual switching element control since all the properties used to derive them apply to each of the networks. Similarly, all the measures for the Generalized Cube network are applicable to all the cube-type networks that have individual switching element control.

The results presented here are for the basic cube-type and data manipulator-type topologies. It should be noted that variations on these topologies which are more fault tolerant have been proposed [2, 12, 27].

The above analysis assumed that the failure of one component was independent of the failure of any other component. If all or a large part of one stage is implemented on a single chip, this assumption may or may not be valid. If it is not, then the networks can be reanalyzed using the techniques presented here to account for the new failure pattern exhibited.

VI. Conclusions

This paper has examined two classes of multistage interconnection networks for use in parallel/distributed systems: the cube type and the data manipulator type. This was done by comparing a representative network from each class: the Generalized Cube and the Augmented Data Manipulator (ADM). This paper has attempted to quantify the differences in implementation costs by considering comparable implementation models for both networks. It was found that a discrete, circuit-switched implementation of the ADM network costs approximately 50% more than the same type of implementation of the Generalized Cube network. For discrete, packet-switched implementations, assuming the packet buffer cost dominates, the two networks cost about the *same* amount (ADM would be slightly higher). If the networks are to be constructed from VLSI chips, assuming the network's building block chips are to have nearly equal numbers of pins, the ADM network requires more than twice as many chips as the Generalized Cube.

Both networks can benefit from VLSI implementation. Each can be partitioned into complex building blocks that have higher logic/pin ratios than partitions of simple building blocks. Though the ADM building block requires more I/O ports on a chip than a Generalized Cube building block,

present and future predicted pin capacities are sufficient for the ADM network's needs. Using bit slicing, arbitrarily wide networks of either type can be constructed.

Using a graph model as a basis, two quantitative measures of comparative robustness were applied to the networks assuming they are used in MIMD or partitioned SIMD environments. Applying the measures to two different (functionally equivalent) implementations of each network under different faults it was found that the ADM network is always more robust. Using the first measure, the Generalized Cube network varied from having 1.14 to 6 times as many affected I/O ports due to a single failure as the ADM network. Using the second measure, in which some I/O ports are disabled, one implementation of the ADM network was shown to be able to fully support communication among the remaining enabled I/O ports.

In summary, a graph model has been used as a basis for quantifying the differences between cube- and data manipulator-type networks. Both implementation costs and robustness have been compared.

APPENDIX: DERIVATIONS OF ROBUSTNESS RESULTS

The following are derivations of each of the results in Tables II and III (excluding those already presented in Section V, B: the average number of affected I/O ports in the Generalized Cube and ADM networks when links or switches fail. Two different implementations and two different rules for disabling I/O ports are considered. Recall that a port is affected by a failure if it cannot send a message to all of the other ports or if it cannot receive a message from all of the other ports.

1. Rule 1: When a link or a switch fails, no I/O ports are disabled.
   A. Implementation: Node = Switch (Arc = Link).
      1. A single link fails. This case was considered in Section V of the text.
      2. A single switch fails. Under this implementation there are $n + 1$ columns of switches (see Figs. 3 and 5) so summations in the average are taken from 0 to n.
         a. Generalized Cube
            Starting at the failed switch, trace links and switches back to the input to determine the number of affected inputs. This number is 2^{n-i} if the failed switch is in column i. Using this same method, but tracing to the output, the number of affected output ports is 2^i. The average number affected is thus

$$\frac{1}{n+1}\sum_{i=0}^{n}(2^{n-i}+2^i)=\frac{2}{n+1}\sum_{i=0}^{n}2^i=\frac{2(2N-1)}{n+1}.$$

b. Augmented Data Manipulator
The same reasoning discussed in the text (Section V) applies here. There are 2^{n-1} affected input ports and only 1 affected output port. The average is thus

$$\frac{1}{n+1}\sum_{i=0}^{n}(2^{n-1}+1) = \frac{1}{n+1}\sum_{i=0}^{n}2^{i}+1$$
$$= \frac{2N-1}{n+1}+1 = \frac{2N+n}{n+1}.$$

B. Implementation: Arc = Switch (Node = Link).
1. A single link fails. This case is completely analogous to I.A.2 above.
2. A single interchange box fails. The effects of this are determined by examining Figs. 1 and 6.
   a. Generalized Cube
   The effects of a failed interchange box in stage i are determined by tracing both input links to the box back to the input and the two output links to the output. The number of affected input ports is 2^{n-i} and output ports is 2^{i+1}. The average number affected is

$$\frac{1}{n}\sum_{i=0}^{n-1}(2^{n-i}+2^{i+1}) = \frac{2}{n}\sum_{i=0}^{n-1}2^{i+1} = \frac{4}{n}\sum_{i=0}^{n-1}2^{i} = \frac{4(N-1)}{n}.$$

   b. Augmented Data Manipulator
   In this implementation, the straight arcs (from Fig. 5) that are paired at stage i (for the network in Fig. 6) are $p_{n-1}\cdots p_{i+1}p_ip_{i-1}\cdots p_0$ and $p_{n-1}\cdots p_{i+1}\bar{p}_ip_{i-1}\cdots p_0$. The logical "distance" between these links is 2^i. Thus, if $j = p_{n-1}\cdots p_{i+1}0p_{i-1}\cdots p_0$ is the address of the upper input to an "interchange" box in stage i, then $j + 2^i = p_{n-1}\cdots p_{i+1}1p_{i-1}\cdots p_0$ is the address of the lower input. For example, in Fig. 6, the second box from the top in stage 1 has inputs with addresses 4 and 6. In binary the addresses are $p_2 0 p_0 = 100$ and $p_2 1 p_0 = 110$, respectively. Notice that each box has two other inputs from nonstraight links. To consider all of the inputs that possibly could be affected by the failure of a box with inputs j and $j + 2^i$ in stage i, trace links backward from each box input to the input of the network. The easiest way to do this is to use Fig. 5. Start with the nodes at levels j and $j + 2^i$ in column i. For the example above, these are

nodes 4 and 6 in column 1. Trace the three links backward to column $i + 1$ and mark the appropriate nodes. Repeat the procedure for each marked node. There will be 2^{n-i-1} inputs marked in column n, so this is the upper bound on the number of affected inputs. For the example, nodes 0, 2, 4, and 6 are marked in column 3 of Figure 5. This translates to inputs with these numbers in Fig. 6. None of the other inputs can be affected by the failure of this box because they have no physical connection to it. All of the marked inputs are affected. This is because any input whose low-order $i + 1$ bits match either j (bits $0p_{i-1} \cdots p_0$) or $j + 2^i$ (bits $1p_{i-1} \cdots p_0$) will require straight connections in stages i through 0 at level j or $j + 2^i$ when these inputs communicate with outputs j or $j + 2^i$. They will be forced to use the faulty interchange box in stage i. Calculation shows that there are 2^{n-i-1} addresses that meet this criterion so the number of affected inputs equals the upper bound.

To determine which outputs are affected requires two observations: (1) inputs that can reach output j or $j + 2^i$ of the faulty interchange box can only get to levels in stage i of the form $j \pm b2^i \bmod N$, b any integer; and (2) regardless of the path taken in stages $n - 1$ though i, when the path reaches the output of stage i, it must be less than a distance of 2^i (i.e., 0 to $2^i - 1$) of the destination D. Observation (1) is a result of the fact that the inputs agree with j in the i low-order bits. In stages $n - 1$ through i the smallest increment by which a path can change levels is 2^i. Thus all the levels it can get to in stage i agree with j in the i low order bits. Observation (2) is a result of the fact that the maximum distance stages $i - 1$ through 0 can change a path is $\sum_{k=0}^{i-1} 2^k = 2^i - 1$.

Now consider five cases regarding the relationship between D, j, and $j + 2^i$. First, note that any interchange box in stage $n - 1$ that fails will affect all the outputs since none of them can receive messages from inputs j and $j + 2^{n-1}$. So, assume $i < n - 1$.

Case 1. $D = j$. Output j is the only output from the faulty box less than a distance of 2 from D, therefore (as shown in the affected inputs analysis) inputs that agree with j in the low-order i bits cannot communicate with D.

Case 2. $D = j + 2^i$. The argument is the same as for Case 1; thus D is affected.

Case 3. $j < D < j + 2^i$. The only outputs in stage i less than 2^i from D are j and $j + 2^i$. Thus both potential paths from an input that agrees with j in the low-order i bits must route through the faulty interchange box, so D is affected.

Case 4. $0 \le D < j$ (if $j = 0 = D$ see Case 1). If D is a distance of 2^{i+1} or more from j, it is completely unaffected because there is no physical path from the faulty box to D. If D is a distance of less than 2^{i+1} from j, the only outputs from the faulty box less than 2^i from D are $j - 2^i$ and j. One of the paths to output $j - 2^i$ comes from the faulty box. However, it is known that there are at least two ways to get from an affected source to output $j - 2^i$ in stage i (which is input $j - 2^i$ of the next stage, stage $i - 1$). This follows from the facts that (1) there are at least two paths between every nonequal network input and output [28], and (2) the only way to reach network output $j - 2^i$ from an affected source is to go through input $j - 2^i$ from stage $i - 1$ and then "straight" through the rest of the network. Therefore, there are at least two physical paths from an affected source to input $j - 2^i$ at stage $i - 1$. Thus, every affected input must be able to communicate with D through the other path to input $j - 2^i$ in stage $i - 1$. Therefore, D is unaffected.

Case 5. $j + 2^i < D \le N - 1$ (if $j + 2^i = N - 1 = D$ see Case 2). This case is completely analogous to Case 4. If D is a distance of 2^{i+1} or more from $j + 2^i$ then it is completely unaffected. Otherwise, the only outputs in stage i less than 2^i from D are $j + 2^i$ and $j + 2^{i+1}$. Thus D is unaffected.

To summarize, if $i = n - 1$, two inputs and all N outputs are affected. For $0 \le i < n - 1$, the outputs affected have an address D such that $j \le D \le j + 2^i$. There are $2^i + 1$ such outputs. The inputs affected agree with j in the low-order i bits. There are 2^{n-i} such inputs. The average number of affected I/O ports is

$$\frac{1}{n}\left[\sum_{i=0}^{n-2}(2^{n-i} + 2^i + 1) + N + 2\right]$$

$$= \frac{1}{n}\left[\sum_{i=0}^{n-2}(4(2^i) + 2^i + 1) + N + 2\right] = \frac{7N - 8}{2n} + 1.$$

II. Rule 2: If a straight link or switching element at level j fails, disable input j and output j. If an interchange box with inputs j and k fails, disable inputs j and k and outputs j and k.

A. Implementation: Node = Switch (Arc = Link).

1. A single link fails.

a. Generalized Cube

When a straight link in stage i fails there are $2^{n+i+1} - 1$ affected inputs and when a nonstraight link fails there are 2^{n+i+1} affected inputs (since no ports are disabled). Similarly there are $2^{i+1} - 1$ and 2^{i+1} affected outputs, respectively. The average is thus

$$\frac{1}{2n}\sum_{i=0}^{n-1}(2^{n+i+1} - 1 + 2^{n+i+1} + 2^{i+1} - 1 + 2^{i+1})$$

$$= \frac{1}{n}\sum_{i=0}^{n-1}(2^{n+i+1} + 2^{i+1}) = \frac{2N-2}{n} - 1.$$

b. Augmented Data Manipulator

If any straight link at level j fails, the only output some of the inputs cannot communicate with is output j. Since it is disabled and it was the only affected output (see the discussion in Section V), none of the remaining enabled input or output ports is affected.

2. A single switch fails.

a. Generalized Cube

This case is similar to case I.A.2.a except that there is one less affected input and one less affected output. Also, the failure of a switch in column n or 0 has no effect on any inputs or outputs. This is because when a column n switch fails, any input other than the one entering that switch can reach all outputs. Similarly, when a column 0 switch fails, the only output that cannot be reached is the one attached to the failed switch. The average is thus

$$\frac{1}{n+1}\sum_{i=1}^{n-1}(2^{n+i+1} + 2^{i+1} - 2) = \frac{2}{n+1}\sum_{i=1}^{n-1}(2^{i+1} - 1)$$

$$= \frac{2N}{n+1} - 2.$$

b. Augmented Data Manipulator

No input or output links are affected. The reasoning is similar to case II.A.1.b. If a switch at level j fails, the

only affected output port is that connected to the faulty switch by straight links, namely, output j.

B. Implementation: Arc = Switch (Node = Link).

1. A single link fails.
   This case is completely analogous to case II.A.2 above.
2. An interchange box fails.
   a. Generalized Cube
      This is similar to case I.B.2.a except that two less inputs and two less outputs are affected. Also, the failure of an interchange box in stage $n-1$ or 0 has no effect on any inputs or outputs. This is because when a stage $n-1$ box fails, any input other than those entering that box can reach all outputs. Similarly, when a stage 0 box fails, the only outputs that cannot be reached are those attached to the failed box. The average is thus

$$\frac{1}{n}\sum_{i=1}^{n-2}(2^{n-i}+2^{i+1}-4)=\frac{2}{n}\sum_{i=1}^{n-2}(2^{i+1}-2)=\frac{2N}{n}-4.$$

   b. Augmented Data Manipulator
      This is similar to case I.B.2.b except that two less inputs and two less outputs are affected. As in case II.B.2.a, the failure of an interchange box in stage $n-1$ or 0 has no effect on any inputs or outputs. Therefore, the average is

$$\frac{1}{n}\sum_{i=1}^{n-2}(2^{n-i}+2^{i}-3)=\frac{1}{n}\sum_{i=1}^{n-2}(3\cdot 2^{i}-3)=\frac{3N}{2n}-3.$$

ACKNOWLEDGMENT

A preliminary version of this material was presented at the Third International Conference on Distributed Computing Systems, October 1982.

REFERENCES

1. Adams, G. B., III, and Siegel, H. J. On the number of permutations performable by the augmented data manipulator network. *IEEE Trans. Comput.* **C-31** (Apr. 1982), 270–277.
2. Adams, G. B., III, and Siegel, H. J. The extra stage cube: A fault-tolerant interconnection network for supersystems. *IEEE Trans. Comput.* **C-31** (May 1982), 443–454.
3. Adams, G. B., III, and Siegel, H. J. The use of 4 × 4 switching elements in the multistage cube networks. Proc. 1st Int. Conf. Computers and Applications, June 1984, pp. 585–592.
4. Agrawal, D. P. Testing and fault-tolerance of multistage interconnection networks. *Computer* **15** (Apr. 1982), 41–53.
5. Agrawal, D. P. Graph theoretical analysis and design of multistage interconnection networks. *IEEE Trans. Comput.* **C-32** (July 1983), 637–648.

6. Anderson, G. A., and Jensen, E. D. Computer interconnection structures: Taxonomy, characteristics, and examples. *ACM Comput. Surveys* **7** (Dec. 1975), 197–213.

7. Barnes, G. H., and Lundstrom, S. F. Design and validation of a connection network for many-processor multiprocessor systems. *Computer* **14** (Dec. 1981), 31–41.

8. Batcher, K. E. STARAN parallel processor system hardware. Proc. AFIPS 1974 Nat. Computer Conf., May 1974, pp. 405–410.

9. Batcher, K. E. The flip network in STARAN. Proc. 1976 Int. Conf. Parallel Processing, Aug. 1976, pp. 65–71.

10. Briggs, F., Fu, K. S., Hwang, K., and Wah, B. W. PUMPS architecture for pattern analysis and image data base management. *IEEE Trans. Comput.* **C-31** (Oct. 1982), 969–983.

11. Cimmiera, L., and Serra, A. Modular interconnection networks with asynchronous control. 14th Annual Hawaii Int. Conf. System Science, Jan. 1981, pp. 210–218.

12. Cimmiera, L., and Serra, A. A fault tolerant connecting network for multiprocessor systems. 1982 Int. Conf. Parallel Processing, Aug. 1982, pp. 113–122.

13. Despain, A. M., and Patterson, D. A. X-tree: A tree structured multi-processor computer architecture. Proc. 5th Annual Int. Symp. Computer Architecture, Apr. 1978, pp. 144–151.

14. Dias, D. M., and Jump, J. R. Analysis and simulation of buffered delta networks. *IEEE Trans. Comput.* **C-30** (Apr. 1981), 273–282.

15. Feng, T. Data manipulating functions in parallel processors and their implementations. *IEEE Trans. Comput.* **C-23** (Mar. 1974), 309–318.

16. Feng, T. A survey of interconnection networks. *Computer* **14** (Dec. 1981), 12–27.

17. Feng, T., and Wu, C. Fault diagnosis for a class of multistage interconnection networks. *IEEE Trans. Comput.* **C-30** (Oct. 1981), 743–758.

18. Flynn, M. J. Very high-speed computing systems. *Proc. IEEE* **54** (Dec. 1966), 1901–1909.

19. Goke, L. R., and Lipovski, G. J. Banyan networks for partitioning multiprocessor systems. Proc. 1st Annual Int. Symp. Computer Architecture, Dec. 1973, pp. 21–28.

20. Jenevein, R. M., and Browne, J. C. A control processor for a reconfigurable array computer. Proc. 9th Annual Int. Symp. Computer Architecture, Apr. 1982, pp. 81–89.

21. Lang, T., and Stone, H. S. A shuffle-exchange network with simplified control. *IEEE Trans. Comput.* **C-25** (Jan. 1976), 55–65.

22. Lawrie, D. H. Access and alignment of data in an array processor. *IEEE Trans. Comput.* **C-24** (Dec. 1975), 1145–1155.

23. Malek, M., and Myre, W. W. A description method of interconnection networks. *IEEE Tech. Committee Distrib. Process. Quart.* **1** (Feb. 1981), 1–6.

24. McDonald, W. C., and Williams, J. M. The advanced data processing test bed. Proc. Compsac, Mar. 1978, pp. 346–351.

25. McMillen, R. J., Adams, G. B., III, and Siegel, H. J. Performance and implementation of 4 × 4 switching nodes in an interconnection network for PASM. Proc. 1981 Int. Conf. Parallel Processing, Aug. 1981, pp. 229–233.

26. McMillen, R. J., and Siegel, H. J. The hybrid cube network. Proc. Distributed Data Acquisition, Computing, and Control Symp., Dec. 1980, pp. 11–22.

27. McMillen, R. J., and Siegel, H. J. Performance and fault tolerance improvements in the inverse augmented data manipulator network. Proc. 9th Annual Int. Symp. Computer Architecture, Apr. 1982, pp. 63–72.

28. McMillen, R. J., and Siegel, H. J. Routing schemes for the augmented data manipulator network in an MIMD system. *IEEE Trans. Comput.* **C-31** (Dec. 1982), 1202–1214.

29. Parker, D. S., and Raghavendra, C. S. The Gamma network. A multiprocessor network with redundant paths. Proc. 9th Annual Int. Symp. Computer Architecture, Apr. 1982, pp. 73–80.

30. Pease, M. C. The indirect binary n-cube microprocessor array. *IEEE Trans. Comput.* **C-26** (May 1977), 458–473.

31. Pradhan, D. K. and Kodandapani, K. L. A uniform representation of single- and multistage interconnection networks used in SIMD machines. *IEEE Trans. Comput.* **C-29** (Sept. 1980), 777–791.

32. Premkumar, U. V., Kapur, R., Malek, M., Lipovski, G. J., and Horne, P. Design and implementation of the banyan interconnection network in TRAC. Proc. AFIPS 1980 Nat. Computer Conf., June 1980, pp. 643–653.

33. Rathi, B. D., and Malek, M. Fault diagnosis of networks. Proc. Distributed Data Acquisition, Computing, and Control Symp., Dec. 1980, pp. 110–119.

34. Sejnowski, M. C., Upchurch, E. T., Kapur, R. N., Charlu, D. P. S., and Lipovski, G. J. An overview of the Texas Reconfigurable Array Computer. Proc. AFIPS 1980 Nat. Computer Conf., June 1980, pp. 631–641.

35. Siegel, H. J. Analysis techniques for SIMD machine interconnection networks and the effects of processor address masks. *IEEE Trans. Comput.* **C-26** (Feb. 1977), 153–161.

36. Siegel, H. J. Interconnection networks for SIMD machines. *Computer* **12** (June 1979), 57–65.

37. Siegel, H. J. *Interconnection Networks for Large-Scale Parallel Processing: Theory and Case Studies*. Lexington Books, Lexington, Mass., 1984.

38. Siegel, H. J., and McMillen R. J. Using the augmented data manipulator network in PASM. *Computer* **14** (Feb. 1981), 25–33.

39. Siegel, H. J., and McMillen R. J. The multistage cube: A versatile interconnection network. *Computer* **14** (Dec. 1981), 65–76.

40. Siegel, H. J., McMillen, R. J., and Mueller, P. T., Jr. A survey of interconnection methods for reconfigurable parallel processing systems. Proc. AFIPS 1979 Nat. Computer Conf., June 1979, pp. 529–542.

41. Siegel, H. J., Siegel, L. J., Kemmerer, F. C., Mueller, P. T., Jr., Smalley, H. E., Jr., and Smith, S. D. PASM: A partitionable SIMD/MIMD system for image processing and pattern recognition. *IEEE Trans. Comput.* **C-30** (Dec. 1981), 934–947.

42. Siegel, H. J., and Smith, S. D. Study of multistage SIMD interconnection networks. Proc. 5th Annual Int. Symp. Computer Architecture, Apr. 1978, pp. 223–229.

43. Smith, S. D. LSI design considerations for multistage interconnection networks for parallel processing systems. Proc. 14th Annual Hawaii Int. Conf. System Science, Jan. 1981, pp. 219–227.

44. Swan, R. J., Fuller, S. H., and Siewiorek, D. P. Cm*: A modular, multimicroprocessor. Proc. AFIPS 1977 Nat. Computer Conf., June 1977, pp. 637–644.

45. Thurber, K. J., Interconnection networks—A survey and assessment. Proc. AFIPS 1974 Nat. Computer Conf., May 1974, pp. 909–919.

46. Tripathi, A. R., and Lipovski, G. J. Packet switching in banyan networks. Proc. 6th Annual Int. Symp. Computer Architecture, Apr. 1979, pp. 160–167.

47. Widdoes, L. C., Jr. The Minerva multi-microprocessor. Proc. 3rd Annual Int. Symp. Computer Architecture, Jan. 1976, pp. 34–39.

48. Wu, C., and Feng, T. On a class of multistage interconnection networks. *IEEE Trans. Comput.* **C-29** (Aug. 1980), 694–702.

49. Wulf, W. A., and Bell, C. G. C.mmp—A multi-miniprocessor. Proc. AFIPS 1972 Fall Joint Computer Conf., Dec. 1972, pp. 765–777.

Paper 9

A Survey of Fault-Tolerant Multistage Networks
and Comparison to the Extra Stage Cube

A SURVEY OF FAULT-TOLERANT MULTISTAGE NETWORKS AND COMPARISON TO THE EXTRA STAGE CUBE

George B. Adams III[†]

Howard Jay Siegel

School of Electrical Engineering
Purdue University
West Lafayette, IN 47907

Abstract

A variety of fault-tolerant multistage interconnection networks for parallel processing systems that have been proposed in the literature are surveyed. A network is fault-tolerant if it can continue to meet its fault tolerance criterion in the presence of one or more failures of the type(s) allowed by its fault model. Significant differences in fault models and fault-tolerance criteria exist among various fault-tolerant networks. This makes direct comparison of these networks difficult. In analyzing the networks, this paper compares the various models and assesses the effect of choosing a common model and criterion. Network characteristics such as degree of fault tolerance, routing control method, and permutation capability are discussed. The networks surveyed and compared to the Extra Stage Cube are the Modified Baseline, Augmented Delta, F-network, Enhanced Inverse Augmented Data Manipulator, Gamma, Fault-Tolerant Benes, and β-networks.

1. Introduction

A number of fault-tolerant multistage interconnection network designs have been discussed in the literature recently. The interconnection network is an important component of large-scale parallel and distributed computer systems since it is the mechanism for information transfer among the computation nodes and memories. Assuring adequate reliability for such large systems is a significant task. Thus, a crucial practical aspect of an interconnection network used to meet communication needs is fault tolerance.

This paper surveys of a number of fault-tolerant multistage interconnection networks which have appeared in the literature. Included are the Extra Stage Cube network [1, 2], the Modified Baseline network [21], the Augmented Delta network [6], the F-network [5], Enhanced Inverse Augmented Data Manipulator [11], the Gamma network [13], the Fault-Tolerant Benes network [3, 17], and β-networks [14]. Only networks with topologies intended to provide fault tolerance are included because an aim of this paper is to compare these networks with the Extra Stage Cube network, the fault tolerance of which is a consequence of its topology. Other methods for enhancing network reliability such as using error correcting codes with existing interconnection networks have been investigated [10] but are not considered here.

Basic terminology is defined in Section 2. Section 3 describes the networks and their characteristics relevant to fault tolerance. The networks are evaluated in Section 4 and compared to the Extra Stage Cube in Section 5.

2. Definitions

Interconnection networks which can continue, in at least some cases, to provide service when they contain faulty components are known as *fault-tolerant*. A network is termed *single fault tolerant* if it can function in spite of a single fault. If up to i faults can be tolerated then the network is *i-fault tolerant*. A network will be termed *robust* if it can tolerate some instances of i faults, but is not i-fault tolerant. A fault is *hard* if it is not of a transient nature. All faults are assumed hard for the purposes of this paper.

It is only meaningful to speak of a network as i-fault tolerant with regard to a particular *fault-tolerance model*. A fault-tolerance model consists of two components. The first, a *fault model*, defines the nature

This research was supported by the U. S. Army Research Office, Department of the Army, under Contract DAAG29-82-K-0101, and the National Science Foundation under Grant ECS 80-16580.

† G. B. Adams III is now with the Research Institute for Advanced Computer Science NASA Ames Research Center, Moffett Field, CA 94035.

of all faults that are assumed to occur in a network. The fault model for a given network may or may not correspond closely to actual or predicted experience with hardware. In particular, fault models are often chosen to have characteristics suited for performing an analysis, even if their characteristics do not exactly reflect reality. The second component is the *fault-tolerance criterion*, the condition that must be met for the network to be said to have tolerated a given fault, or faults. This varies from network to network due to differences in the definition of what constitutes functionality for a given network (basically, what amount of degradation from the fault-free condition is allowed).

The fault tolerance of a network is determined by various factors, including the chosen fault model and fault-tolerance criterion. The choice of fault model, however, is left to the discretion of one investigating the fault tolerance of a network. Different choices can lead to very divergent claims for the fault tolerance of a network. Similarly, various choices of fault-tolerance criteria can imply different fault tolerance capabilities. Thus, the fault-tolerance model is essential to understanding and comparing the fault tolerance capabilities of various networks. Because different fault-tolerance models are used with different networks, some care must be taken when comparing fault-tolerant networks.

Networks are typically all perform their interconnection functions via *stages* of *switching elements* (switches). The number of stages depends on the network, as does the organization of the switching elements. The switching elements are connected by *links*.

3. Network Descriptions and Fault-Tolerance Models

The networks surveyed in this section fall into four general categories. The Extra Stage Cube, Augmented Data Manipulator, Baseline and F-network form a group based on the Generalized Cube network [15, 16]. The networks of these achieve fault tolerance by adding an extra stage of switches to a basic network which is topologically equivalent to the Generalized Cube. The F-network gains fault tolerance by using a Generalized Cube network structure with additional links.

The data manipulator [8] class of networks is represented by the Enhanced Inverse Augmented Data Manipulator (IADM) and Gamma networks. The Enhanced IADM network uses additional links, and the Gamma network uses increased switching element complexity to realize fault tolerance.

The Fault-Tolerant Benes network is a third type of network. It uses $2n-1$ stages of switching elements and an additional switching element to provide fault tolerance compared to the n stages of switches in a Generalized Cube, where $N=2^n$ is the number of inputs. The fourth category of network is represented by β-networks. This family of networks spans a wide range of topologies and gains fault tolerance through an operational technique.

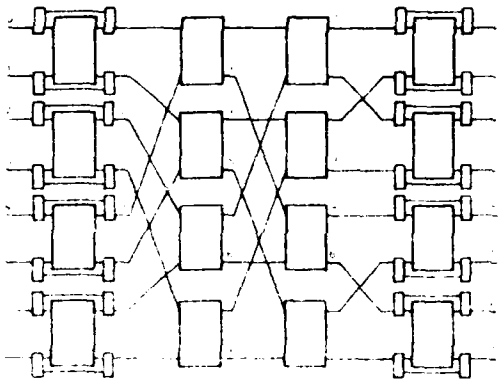


Fig. 1. The Extra Stage Cube network with N=8.

3.1 Extra Stage Cube

The Extra Stage Cube (ESC) [1, 2] is formed from the Generalized Cube by adding an extra stage of switching elements along with a number of multiplexers and demultiplexers. Thus, the ESC has relatively low incremental cost over the Generalized Cube network. ESC network structure is illustrated in Fig. 1 for N=8.

Each stage of the ESC contains N/2 *interchange boxes*, or 2-input/2-output switches. Let the upper input and output lines of an interchange box be labeled i, and the lower lines, j. Then the *straight* setting connects input i to output i and input j to output j. The *exchange* setting connects input i to output j and input j to output i. A *broadcast* connects an input to both interchange box outputs. ESC switching elements are capable of straight and exchange connections and broadcasts from either input to both outputs.

The connections between stages in the ESC are based on the cube interconnection functions [18]. Let $P = p_{n-1} \cdots p_i p_{i-1} \cdots p_1 p_0$ be the binary representation of an arbitrary I/O line label. Then n cube interconnection functions can be defined as

$$cube_i(p_{n-1} \cdots p_1 p_0) = p_{n-1} \cdots p_{i+1} \bar{p}_i p_{i-1} \cdots p_1 p_0$$

where $0 \le i < n$, $0 \le P < N$, and $\bar{p}_i$ denotes the complement of p_i. This means that the cube interconnection *function connects* P to $cube_i(P)$, where $cube_i(P)$ is the I/O line whose label differs from P in just the i^{th} bit position.

Stage i $0 \le i < n$ of the ESC topology contains the $cube_i$ interconnection function, i.e., it pairs I/O lines whose addresses differ in the i^{th} bit position. It is the only stage which can map a source to a destination with an address different from the source address in the i^{th} bit position. When an interchange box in stage i is set to exchange, the data items input to that interchange box are transferred as specified by the $cube_i$ interconnection function. When set to straight, data items input are transferred according to the identity function, where $identity(p_{n-1} \cdots p_0) = p_{n-1} \cdots p_0$. Since each interchange box is individually controlled, each stage i may perform

the $cube_i$ *interconnection function* on all or some subset of the data items depending on the settings of the interchange boxes. The extra stage of the ESC, stage n, is placed on the input side of the network and implements the $cube_0$ *interconnection function*. Thus, there are two stages in the ESC which can perform $cube_0$.

Stage *n* and stage 0 can each be enabled or disabled (bypassed). A stage is ***enabled*** when its interchange boxes are being used to provide interconnection. It is ***disabled*** when its interchange boxes are being bypassed. Enabling and disabling in stages n and 0 is accomplished with a demultiplexer at each box input and a multiplexer at each output. All demultiplexers and multiplexers for stage n share a common control signal, as do those for stage 0. Fig. 2(a) details an interchange box from stage n or 0. The demultiplexer and multiplexer are configured such that they either both connect to their box (enable) or both shunt it (disable) as shown in Fig. 2(b) and 2(c), respectively.

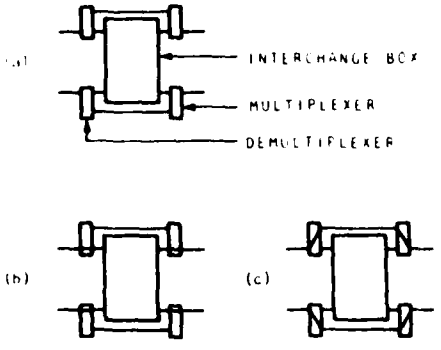


Fig. 2 (a) Detail of interchange box with multiplexer and demultiplexer for enabling and disabling. (b) Interchange box enabled. (c) Interchange box disabled.

Stage n and 0 enabling and disabling is performed by a system control unit. Normally, the network will be set so that stage n is disabled and stage 0 is enabled. If after running fault detection and location tests a fault is found, the ESC is reconfigured. A fault in a stage n box requires no change in network configuration; stage n remains disabled, and the fault isolated. If the fault is in stage 0, stage n is enabled and stage 0 is disabled. Stage n then performs the function of the disabled stage 0. For a fault in any link or in a box in stages n−1 to 1, both stages n and 0 will be enabled. Enabling both stage n and 0 provides tolerance to this type of fault by providing two paths between any source and destination, only one of which can contain the existing fault.

Routing in the ESC is carried out using *routing tags* [9]. Routing tags tags for the ESC, which take full advantage of its fault tolerant capabilities, can be easily computed. The ESC uses n+1 bit routing tags where the i^{th} bit position controls stage i. The routing tag for the fault-free case is given by $T' = t'_n t_{n-1} \ldots t_1 t_0$, where $t_{n-1} \ldots t_1 t_0 = T = S \oplus D$. In the case of faults, bit positions n and 0 of the tag T' may need to be altered, so actual tag values depend on whether the ESC has a fault as well as source and destination addresses, but are readily computed [2]. At each stage i the switching element examines the i^{th} tag bit. If the bit is a 0, the switch is set to straight; if it is a 1, it is set to exchange.

The fault model for the ESC assumes both switching elements and links can fail. However, the input and output ports and the multiplexers and demultiplexers directly connected to the ports of the ESC are always assumed to be functional. If a port or the stage n demultiplexers or stage 0 multiplexers were to be faulty, then the associated device would have no access to the network. The fault-tolerance criterion for the ESC is retention of *full access* capability [5]. Full access capability is the ability to connect any given input to any output. Under its fault model and fault-tolerance criterion the ESC is single fault tolerant and robust in the face of multiple faults [2].

3.2 Modified Baseline Network

The Modified Baseline network [21] is derived from the Baseline multistage interconnection network [20]. The Baseline network has but one path between any source and destination. Thus, any network component failure will affect communication for some set of inputs and outputs. To lessen this difficulty, an extra stage of switching elements is added to the Baseline network. Fig. 3 shows the Modified Baseline network and indicates the original Baseline network and the additional stage. The Modified Baseline network is similar to the ESC except no bypassing of input or output stages is provided. If an extra stage incorporating switching elements with *t* outputs is added at the input side of the network then there are t connection paths between any I/O pair [21].

Routing in the Baseline network is carried out using *destination tags* [9] which consist of the address of the intended destination of a message. If rxr switches are used (r=2 for Fig. 3) then a destination address D can be represented by a base-r number $d_{m-1} \ldots d_1 d_0$ where $m = \log_r N$. This base-r representation is used to select a path through the network in the following way. The switching element connected to the source will use its

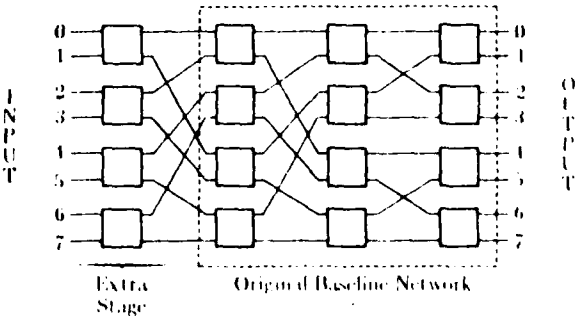


Fig. 3 The Modified Baseline network with N=8

output numbered d_{m-1} to link to a switching element in the next stage. At stage i, d_i is used to determine the selection of switch output, $0 \le i \le m-1$. For the Modified Baseline network an extra digit can be appended to the defined destination tags to control the extra stage.

The fault model for the Modified Baseline assumes only switching elements not in the input or output stages fail. Faulty switches are considered unusable. The fault-tolerance criterion is, as for the ESC, full access. The Modified Baseline network is single fault tolerant and robust in the presence of multiple faults with respect to its fault-tolerance model.

3.3 Augmented Delta Network

An Augmented Delta network [6] is illustrated by Fig. 4.

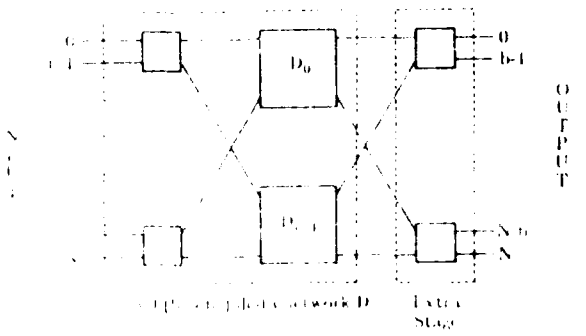


Fig. 4 An Augmented Delta network

[illegible] $\log_b N$. The original Delta network, labeled D [illegible] consists of a stage of N/b bxb switching [illegible] connected to b delta networks, each of size [illegible]. These b networks are labeled D_0 through [illegible] and are each structured in the same [illegible] network.

[illegible] network has a single path between any given [illegible] output. The Augmented Delta network is [illegible] the Modified Baseline network. The distinc[illegible] between the two is that the Augmented Delta net[illegible] may use more than one extra stage and the [illegible] of any extra stage(s) are identical to all others [illegible] [6]. Additional redundant paths can be [illegible] by adding more stages. So that this network [illegible] comparable with the other networks discussed in this paper, only one stage is assumed to be added.

The Augmented Delta network is similar to the ESC. It [illegible] using bxb switching elements, where the ESC [illegible] 2x2 switches, and thus can have more paths [illegible] between a given input and any output. However, like [illegible] Modified Baseline, there is no bypassing of input or [illegible] stages.

The Augmented Delta fault model incorporates the [illegible] assumptions that (1) faults occur in both switching ele[illegible] and links, (2) stage 0 and n switching elements [illegible] always fault-free, (3) faults occur independently, and [illegible] faulty links or switching elements are not available for use. The fault-tolerance criterion is defined as retaining full access capability. Under such a fault-tolerance model, an Augmented Delta network constructed from 2x2 switches is single fault tolerant. If bxb switching elements are used throughout then the network is (b-1)-fault tolerant.

3.4 F-Network

The F-network [5] connects $N=2^n$ inputs to N outputs via n+1 stages of N switching elements which are, in general, 4-input/4-output devices that connect one input to one output. A switching element in stage j, P_j, is denoted by a bit string $P_j=P_{n-1}\ldots P_1P_0$. It is connected to the stage j+1 switching elements $P_{j+1}=P_{n-1}\ldots P_1P_0$, $Q_{j+1}=P_{n-1}\ldots \bar{P}_{j+1}\bar{P}_jP_{j-1}\ldots P_1P_0$, $R_{j+1}=\bar{P}_{n-1}\ldots \bar{P}_{j+1}P_jP_{j-1}\ldots P_1P_0$, and $S_{j+1}=\bar{P}_{n-1}\ldots \bar{P}_{j+1}\bar{P}_jP_{j-1}\ldots P_1P_0$. Fig. 5 shows the F-network for N=8. Stages are numbered from left to

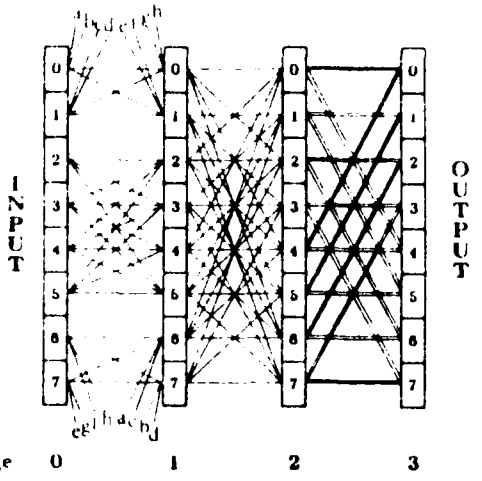


Fig. 5 The F-network for N=8 [5]

right ranging from 0 to n, and within each stage, switching elements are numbered from 0 to N-1.

The F-network contains the structure of the Generalized Cube network and can emulate it using only the P_{j+1} and Q_{j+1} connections. Thus the fault tolerance approach of the F-network is to add links (R_{j+1} and S_{j+1}) to the Generalized Cube structure, unlike the ESC, Modified Baseline and Augmented Delta networks. Routing in the F-network is accomplished through the use of routing tags. The algorithm used to calculate the tags provides for the choice of two of the four output links at any switching element (except for an output stage switch) [5]. This allows the fault-tolerance capabilities of the F-network to be realized.

The fault model used for the F-network assumes (1) faults occur only in switching elements, (2) stage 0 and n switching elements are always fault-free, (3) faults occur independently, and (4) each fault prevents the correct execution of any switching element function, so a faulty switching element is totally unavailable.

The F-network is considered to tolerate faults as long as every input/output pair can communicate. Thus, the fault-tolerance criterion for the F-network is retention of full access. The network is single fault tolerant and robust in the presence of multiple faults with respect to its fault-tolerance model [5].

3.5 Enhanced Inverse Augmented Data Manipulator Network

An Inverse Augmented Data Manipulator (IADM) network is an Augmented Data Manipulator (ADM) network [16] with the order of stage traversal reversed. The ADM is derived from the data manipulator network [8]. Fig. 6 shows the IADM for N=8. It consists of

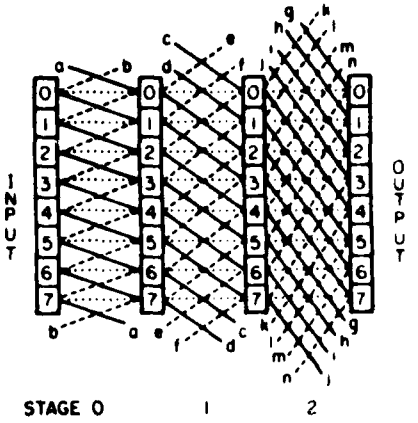


Fig. 6 The Inverse Augmented Data Manipulator network for N=8.

$n = \log_2 N$ stages of N switching elements and 3N links that are connected to the succeeding stage. Each switching element connects one of three inputs to one of three outputs. Specifically, at stage i, $0 \le i < n$, the outputs of switching element j, $0 \le j < N$, are connected to switching elements $(j - 2^i) \bmod N$, j, and $(j + 2^i) \bmod N$ in stage $i + 1$. These links are known as the minus, straight, and plus links, respectively. Since $(j - 2^{n-1})$ is congruent to $(j + 2^{n-1}) \bmod N$, there are actually only two distinct logical data paths from each switching element in stage $n-1$ (stage 2 in Fig. 6). There is an additional set of N switching elements at the output stage.

In [11] performance and fault tolerance enhancements of the IADM are discussed. The fault model for the Enhanced IADM network is the same as for the Augmented Delta network. The criterion for tolerating a fault is also the same.

One method of providing fault tolerance with the IADM is adding redundant straight links. This allows the bypass of a faulty straight link by using the alternate straight link. Faulty plus or minus links can be avoided by taking the alternate path available at the stage just prior to the faulty link [11]. However, switching element faults cannot be tolerated. Routing for the IADM enhanced with straight links is exactly the same as for the IADM network and is performed with routing tags [11].

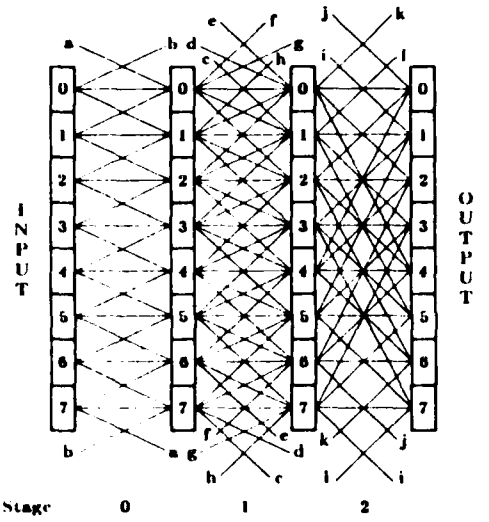


Fig. 7 The Enhanced Inverse Augmented Data Manipulator network with half links for N=8.

A second, more effective modification to gain fault tolerance is to add half links to each of stages 1 through n−1. *Half links* connect a switching element m in stage i to switching elements $(m + 2^{i-1}) \bmod N$ and $(m - 2^{i-1}) \bmod N$. This is shown for N=8 in Fig. 7. Adding half links provides single fault tolerance to any switching element or link failure. This is because at any switching element (except those in stage n−1, the last stage) along a path from a network input to output there are at least two (sometimes three) links leading to distinct switching elements in the successive stage, each of which can be used to satisfy the overall routing need [11]. Again, routing tags as in the IADM can be used. Significant switching element logic is required, however, to interpret the routing tags and allow their dynamic modification to achieve the full fault-tolerance capabilities of the network. This makes these switches candidates for VLSI implementation. With a single-stage look-ahead technique the network becomes two-fault tolerant [11]. That is, messages will not be sent along a route on which all alternative paths to the next stage are blocked by the two faults. Further modifications of the hardware enhancement methods given above are discussed in [11].

3.6 Gamma Network

The Gamma network [13] is adapted from the IADM network (see Fig. 6) and has redundant paths connecting $N = 2^n$ inputs to N outputs. It consists of n stages of N switches. However, unlike the IADM, each of the switching elements is, in general, a 3-input/3-output crossbar switch instead of a one-of-three inputs to one-of-three outputs selector. Switching elements in the input stage have only one input and three outputs, while output stage switches have three inputs and only one output. The connection pattern established by the links is identical to that of the IADM.

In general, the number of paths P_n between an input, or source, S, and an output, or destination, D, in an n stage

Gamma network is

$$P_n(x) = \begin{cases} P_{n-1}[\frac{x}{2} \bmod N], & x \text{ even} \\ P_{n-1}[\frac{x-1}{2} \bmod N] + P_{n-1}[\frac{x+1}{2} \bmod N], & x \text{ odd} \end{cases}$$

where $x = (D-S) \bmod N$, $P_i(0) = 1$, and $P_i(1) = 2$. Note that $P_n(0) = 1$ for all n. The fact that $P_n(x) > 1$ for $x \neq 0$ is the source of fault tolerance in the Gamma network.

The Gamma network can be controlled by n digit routing tags, the value of which is the difference mod N between the numbers of the network input and output to be connected. The digits of the tag may be 1, 0, or -1, corresponding to the $+2^i$, straight, and -2^i links, respectively. Control of the Gamma network when faults occur is not explicitly specified in [13].

A fault model that can be used for the Gamma network assumes (1) faults occur only in switching elements, (2) the input and output stage switching elements are always fault-free, (3) faults occur independently, and (4) faulty switching elements are not available to pass information. The fault-tolerance criterion appropriate for the Gamma network is full access without the stipulation that an input be able to connect to the identically numbered output, as there is only one way to perform this connection. Under this fault-tolerance model the network is single fault tolerant.

3.7 Fault-Tolerant Benes Network

A Benes network [4] connects $N=2^n$ inputs to N outputs with $2n-1$ stages each with $N/2$ 2-input/2-output switching elements. The switching elements can be set to one of two states, straight or exchange. The Benes network is *rearrangeable* in that any idle input/output pair can be connected by rerouting any established one-to-one connections as necessary. In other words, any one-to-one connection can be established regardless of any existing one-to-one connections. Thus, the Benes network is able to form any permutation of inputs to outputs.

Routing in the Fault-Tolerant Benes network is performed by computing the necessary settings of all the switching elements, and then imposing that state on the network through control lines, one per switch. The computation requires time proportional to the number of switches, and faulty switches are not avoided [17]. The required switch settings can be adjusted to match the state of the stuck switch. Faulty switches must be used when permutations are to be performed in only one pass through the network.

The fault model used in [17] for the analysis of fault tolerance of the Benes network is a switching element stuck-fault model. That is, a switching element can be stuck in the straight setting, or stuck in the exchange setting. Further, faults are assumed to occur only in the switching elements of the network, to occur independently, and to be hard. Finally, faulty switching elements are allowed to pass data, and the paths suitably modified to account for the state of the stuck switch. This is a relatively weak fault model in that it supposes an optimistic view of hardware behavior. For example, other switching element failure modes may well be possible, such as ones where continued use of the switching element is not possible. Link failures may also occur in a physical network.

The Benes network can tolerate most single faults, as defined by the above model, where the fault-tolerance criterion is retaining the ability to perform any permutation connection in a single pass through the network. This is also known as *full connection capability* [17]. It is the most stringent fault-tolerance criterion of the networks considered, but the Benes network is the most capable of all the networks considered, in terms of permuting capability. Some multiple switching element faults not in the center stage can be tolerated as well, so the network is robust. However, if any single switching element in the center stage is stuck at the exchange setting then the identity permutation, which connects each input to the identically numbered output, cannot be performed. Also, if any center stage switching element is stuck at the straight setting then the uniform shift connecting each input i, $0 \leq i < N$, to output $N/2 \bmod N$ is not possible.

Any center stage fault can be corrected by a modification that involves adding a single switching element at the input or output stage [17]. The Benes network without modification can tolerate a switching element stuck-at fault at all but the center stage. The addition of the single switching element overcomes this difficulty. The configuration of the fault-tolerant network with the extra switching element at the output is shown in Fig. 8 *for* N=8. Tolerance of a fault is

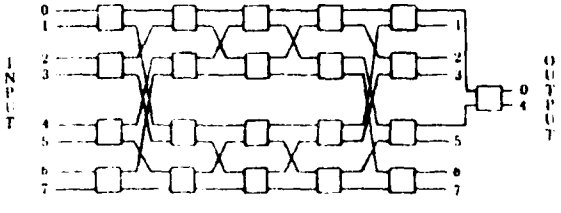


Fig. 8. Fault-tolerant Benes network for N=8.

achieved by using the extra switching element to correct for the misrouting (if any) caused by the fault. Further modifications of the Benes network allowing multiple fault tolerance to switching element stuck-at faults, but requiring extra stages, are described in [17].

3.8 Delta Networks

A *Delta network* is formed by interconnecting a set of $a \times b$ elements [14]. A *Delta element* is a 2×2 crossbar switch which can perform the two permutations exchange and identity (straight). Thus many networks such as the Benes, Baseline, the Modified Baseline, Generalized Cube, and the Extra Stage Cube [2] are examples of Delta networks. An example of a simple Delta network is shown in Fig. 9. Although they may duplicate other surveyed network topologies, Delta networks are studied for their fault model and fault tolerance criterion which differ substantially from those of the other networks con-

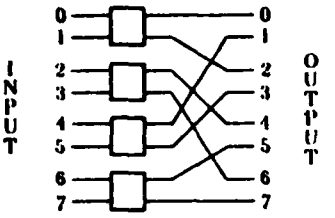


Fig. 9 A simple single-stage β-network.

sidered in this paper. This provides a more complete view of the state of the art in fault-tolerant multistage networks.

A β-network is defined as having the *dynamic full-access* property if each network input can be connected to each network output in a finite number of passes through the network. Between passes it is assumed that each output can connect to its corresponding input (i.e., the input with the same number as the output) via a path outside the network. The β-network is said to tolerate a fault if the fault does not destroy dynamic full-access capability. This is a considerably less restrictive fault-tolerance criterion than is used in any of the other networks surveyed. The purpose in using the dynamic full-access measure is to better characterize the connectivity requirements of computer systems than either full-access or rearrangeability (full connection) capability[14]. However, the multiple pass method of network operation implied by the dynamic full access criterion may be unsuited for some, if not many, applications.

The fault model used for β-networks is the same assumed for the Fault-Tolerant Benes network. Thus, fault tolerance in a β-network is considered to be retention of dynamic full-access using β-elements even with stuck-at faults.

There are two important disadvantages to the β-network approach to fault-tolerant networks. One is the computational complexity of using the dynamic full access criterion. Even when faults have been detected and located considerable work remains to determine the operational status of the network. Specifically, the set of located faults must be tested to see if it comprises a critical fault, one which destroys dynamic full access. The second disadvantage is that by allowing a finite number of passes through the network, data transit time becomes widely variable. This will impose burdens on an SIMD [7] system attempting to maintain synchronization.

Routing in a β-network can be accomplished using binary routing tags with as many bit positions as there are stages in the network. However, β-networks constitute such a broad class that there is no one routing tag scheme generally applicable. Also, realization of dynamic full access capability may incur significant computational expense for routing tags, since a set of tags leading from the original source via a finite number of passes through the network to the ultimate destination must be generated.

3.9 Summary of Network Survey

Table 1 summarizes the network fault tolerance information presented. It gives the possible faults that can occur in each network under the assumed fault model, the fault-tolerance criterion, the method in which the network copes with faults, whether the network is single fault tolerant, and how it performs when there are multiple faults. Note that in the table the phrase "internal node faults only" is another way of saying input and output switching elements are always fault-free.

4. Network Evaluation

There is a growing literature on fault-tolerant multistage interconnection networks. However, as pointed out in [10] many of the results to date have several limitations, including (1) unreasonably optimistic fault models, and (2) increased data routing complexity. As noted earlier, the choice of fault model and fault-tolerance criterion plays a key role in determining the fault tolerance characteristics of a network. In this section the ESC is compared with the other networks surveyed. Table 2 summarizes that comparison. The facts and reasoning supporting Table 2 are discussed below.

ESC fault tolerance is evaluated in light of a fault model that presupposes the possibility of failure of any network component except the stage n demultiplexers and stage 0 multiplexers which are treated as part of the network input/output interface. Stage n multiplexer and stage 0 demultiplexer failures are treated as stage n and stage 1 link failures, respectively. As can be seen from Tables 1 and 2, this fault model is stricter than the fault models of the comparison networks. That is, it assumes at least as many possibilities for failure as the other models (both switching elements and links) and dire consequences for such failures (any faulty component is unusable). The ESC fault model may well be the most realistic of these fault models.

The fault-tolerance criterion for the ESC is the same as that for most of the networks surveyed. Basically, what is required is that one-to-one interconnection capability be uncompromised. The Fault-Tolerant Benes network uses the more demanding criterion that permuting capability be unaffected: all permutations should still be performable with a single pass through the network. It is appropriate to use this strict criterion because the Fault-Tolerant Benes network, unlike the other networks considered, is capable of full connection capability.

The fault-tolerance criterion used to study β-networks is a much less strict test to pass. All that is required is that it be possible to connect any input to any output in a finite number of passes through the network. Successive passes are performed by returning data from a network output to the same numbered input. In a fault-free condition a β-network may require multiple passes for data to reach its destination, so the chosen fault-tolerance criterion is appropriate. However, since the class of β-networks is so broad, it is important to note that this forgiving criterion may inflate the capabilities attributed to more complex β-networks. The fault-

Table 1 Summary of fault tolerance information for the networks surveyed. "SE" is an abbreviation for switching element.

Network	Fault Model	Fault Tolerance Criterion	Fault Tolerance Method	Single Fault Tolerant*	Multiple Fault Tolerance
Extra Stage Cube	any SE or link unusable	full access	alternate route	yes	robust
Modified Baseline	internal SE only; unusable	full access	alternate route	yes	robust
Augmented Delta	internal SE or link, unusable	full access	alternate route	yes	robust, (b−1)-fault tolerant with b×b switches
F-network	internal SE only, unusable	full access	alternate route	yes	robust
Enhanced IADM (straight links)	internal SE or link unusable	full access	alternate route	no	robust
Enhanced IADM (half links)	internal SE or link; unusable	full access	alternate route	yes	robust, 2-fault tolerant with lookahead
Gamma	internal SE only; unusable	full access, but no identity connection	alternate route	yes	robust
Fault-Tolerant Benes	SE stuck, but usable	full connection capability	correct misroute	yes	robust
β-networks	SE stuck, but usable	dynamic full access	repeated pass	depends on network	typically robust

* Answer depends critically on fault model and fault-tolerance criterion.

Table 2 Comparison of the fault tolerant interconnection networks with the ESC network. Entries in the table give the relationship between the network in question and the ESC as regards a particular attribute.

Network	Fault Model	Fault Tolerance Criterion	Routing Complexity	Fault Tolerance Capability*
Modified Baseline	less strict	same	similar	similar
Augmented Delta	slightly less strict	same	similar	similar
F-network	less strict	same	similar	similar
Enhanced IADM (straight links)	slightly less strict	same	similar	less
Enhanced IADM (half links)	slightly less strict	same	similar, complexity hidden in switching element logic	greater if more complex routing implemented
Gamma	less strict	same	similar	similar
Fault-Tolerant Benes	much less strict	stricter, network more capable	much higher	similar
β-networks	much less strict	much less strict	much higher	similar to greater, depends on network

* Using the fault model and fault-tolerance criterion defined for that network.

tolerance criterion of the Gamma network may be unsuitable for some computer systems as it does not consider inability to connect an input to the identically numbered output (an identity connection) to be a failure. If the same device, e.g., a processor-memory pair, is connected to the same numbered input and output then this is not a problem, since a device should not need to communicate with itself.

For most of the networks, routing in the presence of faults is little more complex than in the absence of faults. The notable exception to this is δ-networks. The dynamic full access procedure requires chosing a set of intermediate outputs which can each be reached consecutively, such that the ultimate destination can be reached in one pass from the input with the same number (address) as the last intermediate output. A general solution to this problem is not known. Routing complexity for the Fault-Tolerant Benes network is higher than for the ESC because of the nature of the Benes network [12]. It is not due to the modification for fault tolerance.

5. Comparison to the Extra Stage Cube

The fault tolerance capabilities of the networks considered are all reasonably similar given the various bases by which they are determined. This is apparent from the column on fault tolerance capabilities in Table 2. There should be no surprise that this is so. It is easy to agree with the idea that a network should have whatever fault tolerance capabilities are feasible, and single fault tolerance is more feasible than i-fault tolerance, $i > 1$. However, because each network is studied using its own fault-tolerance model significant differences in capabilities might appear if a common fault model is adopted.

The ESC fault model and fault-tolerance criterion can be applied to the other surveyed networks in order to relate their fault tolerance to that of the ESC. This information is given in the first column of Table 3. Under the ESC fault model and fault-tolerance criterion none of the surveyed networks is single fault tolerant. Many of the networks fail to be single fault tolerant because they cannot tolerate an input or output switching element fault, as can the ESC. This is why so many of the fault models refer only to internal switching element faults. If the ESC fault model is amended to assume fault-free switching elements in the input and output stages, some of the networks become single fault tolerant as shown in the table.

The Fault-Tolerant Benes network is capable of single fault tolerant operation under the relaxed ESC fault model. Although faulty components cannot be used to pass data under the ESC fault model (unlike the Fault-Tolerant Benes fault model), only one-to-one connections need be supported (as compared to permutation connections for the Fault-Tolerant Benes fault model). The Fault-Tolerant Benes network can perform any one-to-one connection without using a given faulty component. However, the control method given in [17] must be modified to achieve this fault tolerance capability so that faulty network components are avoided (the given algorithm uses faulty components).

Table 3 Fault tolerance capabilities of the networks using the ESC network fault model and fault tolerance criterion.

A Single fault tolerant using ESC fault model and fault-tolerance criterion.

B Single fault tolerant if ESC fault model is relaxed to assume input and output stage switching elements are fault-free.

Network	A	B
Extra Stage Cube	yes	yes
Modified Baseline	no	yes
Augmented Delta	no	yes
F-network	no	yes
Enhanced IADM (straight links)	no	no
Enhanced IADM (half links)	no	yes
Gamma	no	no
Fault-Tolerant Benes	no	yes†
δ-networks	no	yes*

†*Must modify control* scheme to achieve fault tolerance under this model.

*Typically yes, but depends on network.

The Enhanced IADM with redundant straight links is not single fault tolerant when the ESC fault model is relaxed because it still cannot tolerate all switching element failures. This includes the switching element failures in interior stages allowed under the modified fault model. The additional straight links provide fault tolerance against the loss of a straight link, but not a switch. The fault-tolerance capability with respect to switches is the same as the IADM, and there are many cases where a switch failure will block a connection [11]. The Gamma network is not single fault tolerant under the relaxed ESC fault-tolerance model because it has only one path from an input to the identically numbered output. A straight-link fault will prevent an input from communicating with the identically numbered output (as it would in the IADM network on which the Gamma network is based). Thus, the Gamma network does not satisfy the ESC fault-tolerance criterion of full access.

6. Conclusions

Eight fault-tolerant interconnection networks have been described. All have multiple stages, but there are wide variations in topology and switching element design.

The Fault-Tolerant Benes network and β-networks are both composed of β-elements, but have differing link patterns. The Gamma network uses 3x3 crossbar switching elements and the same link connection pattern as the ADM. The enhanced IADM and F-network use 5x5 and 4x4 switches, respectively, which pass one item at a time. The Augmented Delta, Modified Baseline, and ESC networks are all cube-type networks [19], and each incorporates an extra stage of switching elements to provide redundant paths. The ESC provides for bypassing of faulty input and output stage switching elements.

Despite its challenging fault model and fault-tolerance criterion, the ESC network is single fault tolerant. The fault model and fault-tolerance criterion for the ESC were chosen for their consistency with engineering practice. The ESC is intended to be a viable answer to real parallel computer interconnection needs.

References

[1] G. B. Adams III and H. J. Siegel, "A multistage network with an additional stage for fault tolerance," *1982 Hawaii Conf. System Sciences*, Jan. 1982, pp. 538-542.

[2] G. B. Adams III and H. J. Siegel, "The extra stage cube: A fault-tolerant interconnection network for supersystems," *IEEE Trans. Computers*, vol. C-31, pp. 443-454, May 1982.

[3] D. P. Agrawal, "Testing and fault tolerance of multistage interconnection networks," *Computer*, vol. 15, pp. 41-53, Apr. 1982.

[4] V. E. Benes, *Mathematical Theory of Connecting Networks and Telephone Traffic*, Academic Press, New York, 1965.

[5] L. Cimmiera and A. Serra, "A fault-tolerant connecting network for multiprocessor systems," *1982 Int'l Conf. Parallel Processing*, Aug. 1982, pp. 113-122.

[6] B. M. Dias and J. R. Jump, "Augmented and pruned N log N multistage networks: Topology and performance," *1982 Int'l Conf. Parallel Processing*, Aug. 1982, pp. 10-11.

[7] M. J. Flynn, "Very high-speed computing systems," *Proc. IEEE*, vol. 54, pp. 1901-1909, Dec. 1966.

[8] T. Y. Feng, "Data manipulating functions in parallel processors and their implementations," *IEEE Trans. Computers*, vol. C-23, pp. 309-318, Mar. 1974.

[9] D. H. Lawrie, "Access and alignment of data in an array processor," *IEEE Trans. Computers*, vol. C-24, pp. 1145-1155, Dec. 1975.

[10] J. F. Lilienkamp, D. H. Lawrie, and P. C. Yew, "A fault tolerant interconnection network using error correcting codes," *1982 Int'l Conf. Parallel Processing*, Aug. 1982, pp. 123-125.

[11] R. J. McMillen and H. J. Siegel, "Performance and fault tolerance improvements in the inverse augmented data manipulator network," *9th Symp. Computer Architecture*, Apr. 1982, pp. 63-72.

[12] D. C. Opferman and N. T. Tsao-Wu, "On a class of rearrangeable switching networks - Part I: Control algorithm," *Bell System Technical Journ.*, vol. 50, pp. 1601-1618, May-June 1971.

[13] D. S. Parker and C. S. Raghavendra, "The gamma network: A multiprocessor interconnection network with redundant paths," *9th Symp. Computer Architecture*, Apr. 1982, pp. 73-80.

[14] J. P. Shen and J. P. Hayes, "Fault tolerance of a class of connecting networks," *7th Symp. Computer Architecture*, May 1980, pp. 61-71.

[15] H. J. Siegel and R. J. McMillen, "The multistage cube: A versatile interconnection network," *Computer*, vol. 14, pp. 65-76, Dec. 1981.

[16] H. J. Siegel and S. D. Smith, "Study of multistage SIMD interconnection networks," *5th Symp. Computer Architecture*, Apr. 1978, pp. 223-229.

[17] S. Sowrirajan and S. M. Reddy, "A design for fault-tolerant full connection networks," *1980 Conf. Info. Sciences and Systems*, Princeton Univ., Mar. 1980, pp. 536-540.

[18] H. J. Siegel, "Analysis techniques for SIMD machine interconnection networks and the effects of processor address masks," *IEEE Trans. Computers*, vol. C-26, pp. 153-161, Feb. 1977.

[19] H. J. Siegel, *Interconnection Networks for Large Scale Parallel Processing: Theory and Case Studies*, D. C. Heath and Company, Lexington, MA, to be published, 1983.

[20] C. L. Wu and T. Y. Feng, "On a class of multistage interconnection networks," *IEEE Trans. Computers*, vol. C-29, pp. 694-702, Aug. 1980.

[21] C. L. Wu, T. Y. Feng, and M. C. Lin, "Star: A local network system for real-time management of imagery data," *IEEE Trans. Computers*, vol. C-31, pp. 923-933, Oct. 1982.

Biographies

George B. Adams III joined the Research Institute for Advanced Computer Science (RIACS) in August 1983 where he is a Research Engineer. He is completing the requirements for the Ph.D. degree from the School of Electrical Engineering, Purdue University. He received the M.S.E.E. degree from Purdue University in 1980 and the B.S.E.E. degree from Virginia Polytechnic Institute and State University in 1978. His research interests include interconnection network design and analysis, computer architecture, and concurrent processing algorithms. Mr. Adams is a member of the IEEE and the Eta Kappa Nu, Tau Beta Pi, and Phi Kappa Phi honorary societies.

Howard Jay Siegel received the S.B. degree in electrical engineering and the S.B. degree in management from the Massachusetts Institute of Technology, Cambridge, MA, in 1972, the M.A. and M.S.E. degrees in 1974, and the Ph.D. degree in 1977, all in electrical engineering and computer science from Princeton University, Princeton, NJ. In 1976 Dr. Siegel joined the School of Electrical Engineering, Purdue University, West Lafayette, IN, where he is currently an Associate Professor. His research interests include parallel/distributed processing, multimicroprocessor systems, image processing, and speech processing. Dr. Siegel is the Chairman of the ACM SIGARCH (Special Interest Group on Computer Architecture) and is a member of the Eta Kappa Nu and Sigma Xi honorary societies.

Paper 10

The LOCO Approach to Distributed Task Allocation in AIDA by VERDI

From the Proceedings of The 5th International Conference on Distributed Computing Systems, May 1985.

THE LOCO APPROACH TO DISTRIBUTED TASK ALLOCATION IN AIDA BY VERDI

V. M. Milutinović, J. J* Crnković, L.-Y. Chang, and H. J. Siegel
School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907
(317)-494-3530

ABSTRACT

A system of special purpose processing resources shared by a number of general purpose processing resources is considered. We assume that special purpose processing resources are dedicated to different tasks typical of complex artificial intelligence multitask jobs. Possible types of special purpose processing resources include pipelined array processors, SIMD parallel processor systems, or MIMD multiprocessor systems, with associated data bases or knowledge bases, for numeric or symbolic computing. Each specific type may be represented by several units. Such a structure may be found in the large local area networks of the 1990s which are used predominantly for artificial intelligence, or in high-end computers of the 5th generation. Given such a processing environment, in this paper an approach for efficient distributed task allocation is introduced. It is referred to as the LOCO approach, because an analogy with a locomotive engine (and appended wagons) is used to describe it. An analytic model of the LOCO approach is developed and used for performance analysis. Results of the performance analysis are presented comparatively with those of load balancing applied to the same processing environment. Although our primary concern is a processing environment for artificial intelligence, we find that the LOCO approach can be used efficiently in other types of processing environments, as well.

KEYWORDS: *Distributed Task Allocation, Loosely Coupled Multiprocessor Systems, Artificial Intelligence Oriented Systems.*

I. INTRODUCTION

Conventional general purpose (*GP*) computers are not able to meet the complex computational requirements typical of artificial intelligence (*AI*). However, the overall processing power of the conventional GP computers can be considerably enhanced if appropriate special purpose processing resources (*SPPRs*) are attached to them. In that case, the GP computers serve as the hosts and the SPPRs as their computational enhancements. In such a processing environment, the SPPRs are oriented to various specialized tasks typical of complex multitask AI jobs. These tasks may be signal processing [e.g., RabGo75], natural language processing/understanding [e.g., Grosz82], vision processing/understanding [e.g., Brady82], intelligent retrieval from knowledge bases within the expert systems [e.g., HaWaL83], etc. Internally, the SPPRs may be organized as special function processors, systolic arrays, pipelined array processors, SIMD machines [Flynn72], or MIMD machines [Flynn72]. Each SPPR will have an associated data base or knowledge base, and will be oriented to numeric or symbolic processing. We have many examples of internal organizations oriented to dedicated control [e.g., MilWa83, Milut83], signal processing [e.g., McDMa82], speech processing/understanding [e.g., Lerne80], image processing/understanding [e.g., SiSiK81], efficient retrieval from relational data bases [e.g., MuKaM83], combinatorial search [WahMa84], inference [Ushid83, SuHoS81], etc. A single SPPR may consist of a large number of processing elements (*PEs*). As an example, the MPP processor for image processing [Batch80] includes 2^{14} PEs. The DADO production system is supposed to include on the order of magnitude of a hundred thousand PEs [StoSh82]. A large number of PEs is used to speed up a special-purpose computation, and not to acquire general processing power.

As the application requirements are getting more and more complex, the overall computational capabilities can be further increased by adding more SPPRs to a GP host. Several experiments of this type were reported [MarBr81]. This trend is very likely to continue, especially given the great importance and massive computational requirements typical of the general area of AI. Consequently we expect that in the 1990s systems oriented to AI will consist of hundreds of SPPRs. As the SPPRs will still be costly, it will make sense to share them among a number of GP hosts. It is very unlikely that all SPPRs in such a system will be of the same type, and it is even more unlikely that each one will be different from the others. We expect that they will be of a variety of different types where each type is represented by an appropriate number of units. Different SPPRs will be of different levels of specialization. It is reasonable to expect that the whole system of GP hosts and SPPRs will span an area the size of a typical university campus or military base (up to about 1 mile in radius).

A similar type of processing environment may also be found in high-end computers of the 5th generation. Input/output (in the wide sense) will include natural language processing and computer vision. Memory (in the wide sense) will incorporate a variety of data bases and knowledge bases. The processor (in the wide sense) will incorporate the capabilities for both information and knowledge processing [TreLi82].

In both cases it will be extremely important to have an efficient mechanism for the dynamic allocation of different tasks belonging to complex AI jobs [Davis83]. For a number of reasons, this mechanism must be distributed in nature. It might exist as distinct and identifiable blocks of code, or only as a design philosophy [Enslo78, JenPl84]. The complexity of the problem is higher than what may initially be expected, as most of the hosts may be working in a multiprogramming environment, where different processes running on the same host will have jobs with tasks oriented to different SPPRs. Also, the allocation requirements will change in time. Obviously, the solution of this problem should involve the following two basic aspects:
(a) System architecture that supports an efficient task allocation
(b) Dynamic task allocation procedure which is distributed in nature.
With all that in mind, in this paper a system architecture is considered which consists of GP hosts and logically clustered SPPRs, all connected by a shared multiple access bus, possibly but not necessarily of the CSMA/CD type [e.g., ShDaR82]. Such a structure is well suited to the execution of complex multitask jobs typical of AI and will be referred to as *AIDA* (Artificial Intelligence Directed Architecture). For this type of system architecture an efficient approach to dynamic and distributed task allocation is introduced and analyzed. It is referred to as the *LOCO* approach, because an analogy with a locomotive engine (and appended wagons) is used to describe it [MilSi84]. As will be seen later, this approach is quite general in nature, and can be applied to processing environments other than the one described here.

It should be noted that the emphasis here is on a system that consists of a large number of heterogeneous classes of SPPRs, made accessible to a very large number of users through a large number of hosts. Furthermore, due to the variety of SPPRs, the types of computations to be performed are unlimited -- the system is not restricted to any single task domain. This makes it very appropriate for environments, such as AI, that require many different types of tasks to be executed. This can be contrasted to a parallel processing system like PASM [SiSiK81] in the following ways: (1) PASM's computation engine consists of a set of homogeneous processors, (2) the PASM processors are interconnected by a multistage network [Siege84], rather than a network of shared busses, (3) when operating in SIMD mode, the PASM processors exploit instruction level parallelism, while the inter-SPPR parallelism is on the task level, (4) PASM is intended for image understanding [KuSiA84], where LOCO/AIDA is much more "general purpose" in nature, (5) PASM is intended to support a much smaller set of

*J. J. Crnković is now with the Department of Mathematical and Computer Sciences, University of Miami, Coral Gables, Florida 33124.

This research was partially supported by the U.S. Army Research Office, Department of the Army, under contract DAAG29-82-K0101, and the School of Electrical Engineering, Purdue University.

Proceedings of the 5th IEEE International Conference on Distributed Computing Systems, Denver, Colorado, May 13-17, 1985.

users than LOCO/AIDA and (6) PASM itself could be an SPPR in LOCO/AIDA.

One excellent study of distributed resource sharing is given in [Wah84]. According to it, AIDA can be treated as a resource sharing network architecture based on a single shared bus, and LOCO as a procedure with an addressing mechanism distributed in the network. Also, the LOCO/AIDA environment is characterized by a number of elements typical of the data flow environment [GaPaK82].

This paper is organized into six sections. Assumptions of the analysis are addressed in Section II. The system architecture (AIDA) is introduced in Section III. The distributed procedure for dynamic task allocation (LOCO) is introduced in Section IV, first through an example and then generalized. A model of the LOCO approach based on AIDA architecture is introduced in Section V. For comparison purposes, in the same section, a model of load balancing (*LB*) applied to the same system architecture is introduced, as well. Performance analysis of the LOCO approach and its comparison to the LB approach are given in Section VI.

II. BASIC ASSUMPTIONS OF THE ANALYSIS

The presentation to follow will be based on the following assumptions:

1. A task is the monolithic complex of computation that can be performed by an SPPR without any intermediate interaction by the host. In other words, once an SPPR is loaded with the program, its parameters and the data, it can autonomously execute the task until its completion. The tasks are highly specialized. Consequently, a great variety of different SPPR types is necessary. For the number of different SPPR types (N_T) we assume $N_T >> 1$.

2. Each task (T) is part of the job (J) that belongs to a process (P) running in one of the hosts (H). A number of processes can run concurrently in the same host. A single process may include a number of jobs (running sequentially or concurrently). A single job consists of a number of different tasks (running sequentially or concurrently). This fact can be symbolically represented as in the example of Fig. 1. Con-

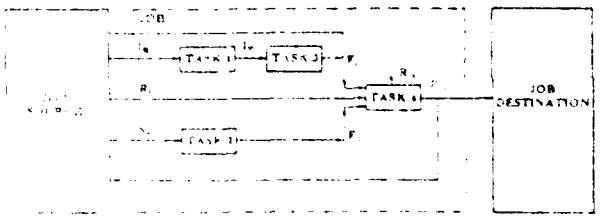


Fig. 1. Example of a Complex Multitask AI Job

[illegible] Input image — F_{in} – Signal features
[illegible] Transformed image — R_{in} – Input rules
[illegible] Image features — R_{bi} – Built-in rules
[illegible] — P_{out} – Output predicates

[illegible] represent the inter-task data dependency. In the example of Fig. 1, TASK 1 and TASK 2 can run concurrently on two different [illegible]; while TASK 3 must wait for TASK 1 to be completed, since it needs TASK 1's output data. The same task may exist in various contexts in different jobs. As the number of jobs can be very large, it may help if each job type is represented by more than one unit. For the number of instances per job type (N_J) we assume $N_J >> 1$.

The total number of SPPRs in the system (N_S) is given by [illegible]. We anticipate that in real systems of the 1990s this number may get to be on the order of several hundreds. The number of hosts acting as job generators is assumed to be of the same order of magnitude. These facts justify the use of an infinite population model in the analysis to follow.

As stated earlier, the SPPRs may be more or less specialized. A specific task can run on one SPPR type only, or on one of a number of different SPPR types. In the second case, however, one of the SPPR types will be the most suitable. In the presentation to follow, a task will always be associated with one SPPR type, regardless of if it is the only possibility, or the most suitable possibility. This assumption simplifies the presentation without affecting its generality.

[illegible] Unless otherwise noted, we assume that each SPPR type is represented by the same number of units. This will simplify the notation without affecting the generality of our analysis.

5. The duration of a task execution is considerably longer than the time needed to transfer data to the SPPR that will execute the task. The transfer time includes the time to access the communications medium and to exchange control data. This assumption is quite realistic. On one hand, advanced fiber optic technology is enabling local area communications to reach gigabit/second speeds [e.g., PoCoS83]. On the other hand, the execution time of AI tasks may be extremely high. This is due to extensive data quantities (tasks with 2^{16} pixels and/or 2^{10} logic rules are not uncommon) and extensive computational intensity (for both numeric processing and logic search). As known from previous work [e.g., Wah84], if the task transmission time is small compared to the task service time, the single bus approach is the best approach. This was the justification for us to concentrate in our research on the single bus system architecture for support of the task allocation.

6. Each task in execution can be treated as a secondary process (running on the SPPR) that can generate a number of secondary jobs, each one consisting of a number of secondary tasks. The nesting can continue as necessary. We treat this issue as a VERtically Distributed Intertasking, or simply VERDI. Consequently, the system architecture is referred to as the AIDA by VERDI. We mention this nesting as an interesting property of the LOCO approach. However, that issue will not be further analyzed in this work.

7. AI tasks (both those predominantly oriented to numeric and symbolic processing) are characterized by large execution time variations [Brady82, Grosz82]. The same conclusion has been derived by a recent study [Rober84]. Consequently, the correlation between past experience on execution time for a given type of task and its future execution time is low. This fact represents one of the essential differences between typical AI tasks, and the tasks typical of the "conventional" *GP* processing environment. *It is of crucial importance for the analysis to follow.* As will be seen later, this fact has a major influence on our choice of the task allocation procedure, and the underlying system architecture.

8. Programming of the SPPRs is very complex. Although the user can develop its own software, typically parametric library routines are used. The user's major effort is to specify the software parameters. The library routines are assumed to be relatively short, as they control specialized processing resources. These facts influence the choice of the system architecture for the efficient support of task allocation.

9. The area over which the system is spanned, as well as the bandwidth of the communications medium, ensure that the propagation time between any two points in the system is negligible in comparison with duration of the shortest possible message. We assume that enough bandwidth is available, so that computation and not the communication is a system bottleneck. This assumption permits us to neglect the media access and handshaking effects in the analysis to follow.

10. This paper concentrates on the case when SPPRs are the bottleneck in the system. The case when both SPPRs and interconnection network are the bottleneck in the system is not considered as a part of this work.

Unless otherwise noted, all these assumptions will be used throughout the presentation to follow.

III. SYSTEM ARCHITECTURE

One possible approach to a system architecture for the processing environment treated here implies connecting of SPPRs to the back-ends of the hosts and connecting of hosts into a single shared bus network, as indicated in Fig. 2a. This approach permits load balancing [e.g., HwCrG82], but reliability and expandability may be problematic. We follow here another approach according to which the SPPRs are moved into the front-end and share the same bus with the hosts, as indicated in Fig. 2b. This approach has good reliability and expandability. It supports load balancing and several good papers exist on that topic [e.g., WahJu83, WahHi82]. Load balancing is very efficient if execution times of the tasks waiting for processing can be precisely estimated. This estimation may not be possible or may require intensive computation [FanTo84]. So allocation typically relies on past experience about the execution time for a given type of task. If the correlation between past and future execution time is relatively high, the load balancing proves to achieve a very good performance [e.g., NdHwA81, ChoKo79]. Unfortunately, this assumption is not satisfied in our case and we cannot use the existing results. We were forced to search for appropriate task allocation procedure and a system architecture that are efficient without any *knowledge about task execution times*.

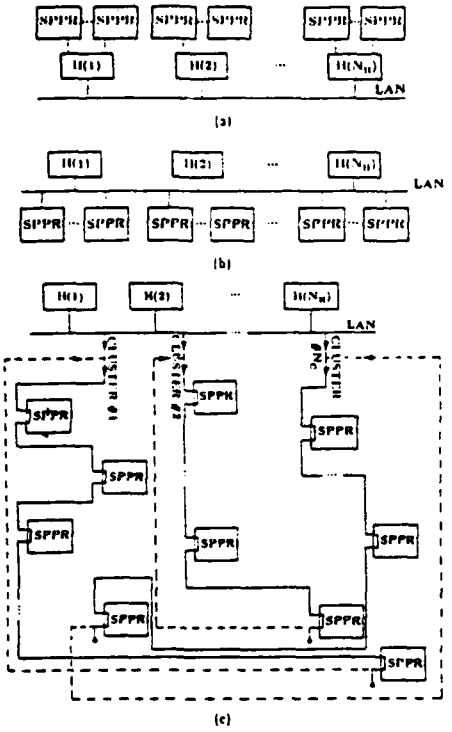


Figure 2. *Some Possible System Architectures for Distributed Processing on a LAN (Local Area Network).*

(a) The SPPRs in back-end of the hosts, with load balancing.
(b) The SPPRs directly appended to the LAN, with load balancing.
(c) Logical clustering of physically remote SPPRs. The SPPRs are assumed to be physically remote. However, they are connected by a high-speed link, and they behave as if they were locally clustered, *i.e.*, logically clustered.

Fortunately, *the existence of a high-speed communications medium gives an important new dimension to distributed processing.* It enables the introduction of the concept of logical clustering of physically remote SPPRs of the same type. Consequently, given a fast enough communications medium and a small enough local area, it makes sense to interconnect all SPPRs of one type into a single logical cluster, as indicated in Fig. 2c. Although physically remote, these SPPRs behave as if they were logically local to each other. Consequently, we have N_C logical clusters, with N_U SPPRs of the same type in each.

Logical clustering means that all SPPRs of the same type are treated as a single multiple-server service station. No load balancing is needed any more as it implies the environment characterized by multiple single-server service stations. Also, the lack of correlation between past and future is of no importance any more. The task is simply sent to a logical cluster that consists of all SPPRs which are best suited to its efficient execution. It waits in the queue associated to the logical cluster until after all the previously arrived (in the case of FIFO disciplines) or higher priority tasks (in the case of priority disciplines) have been served. Note that the concept of clusters in our approach is considerably different compared to the concept of clusters in Cm* [SwFuS77], Ultracomputer [GoGrK83], or Cedar [GaLaK84].

Now we will describe the Artificial Intelligence Directed Architecture (AIDA), which is based on the above described principle of logic clustering. It is given in Fig. 3. The system consists of N_H hosts and N_S SPPRs organized into N_C clusters with N_U SPPRs per cluster. Each cluster is associated with a mass storage unit M(j); j=1,....N_C. This is where the software (parametric library routines) for all SPPRs

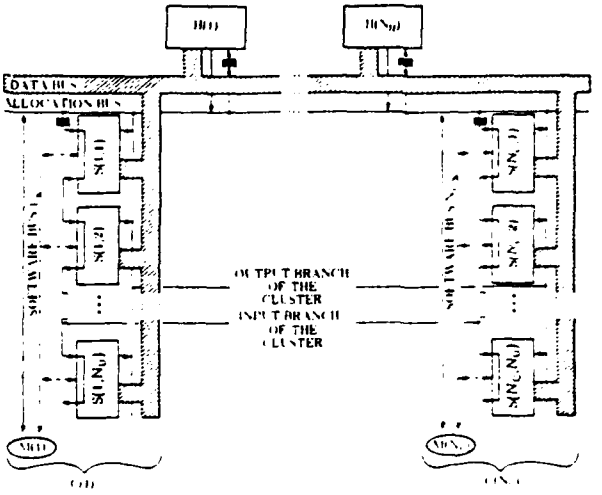


Figure 3. The AIDA: An Architecture for Efficient Support of the LOCO Approach to Distributed Task Allocation.

in that cluster is stored. Knowledge bases and data bases can exist within SPPRs, in the mass storage units associated with the cluster, or in any other suitable form. The SPPRs are interconnected by a system of buses. Separate buses are used for task allocation, for data and parameter transfer, and for the transfer of the library routines. These buses will be referred to as the allocation, data, and software bus, respectively. The allocation bus is a single-line bus (bit transfer). It connects the hosts, the SPPRs, and the mass storage units, as the software libraries have to be updated occasionally. It includes the cluster branches. Each cluster branch is separated into the INPUT.BRANCH and the OUTPUT.BRANCH. The INPUT.BRANCH is daisy chained, as indicated in Fig. 3. Given assumption #8, the software bus can also be a single-line bus (bit transfer). We assume one software bus per logical cluster. The software buses connect the SPPRs of the cluster with the corresponding mass storage unit. Given assumption #5, the data bus should be a multiple-line bus (word transfer). It connects the hosts and the SPPRs. An identification number (*ID*) is assigned to each host (*H.ID*), process (*P.ID*), job (*J.ID*) and task (*T.ID*). Identification numbers are also associated to the clusters (*C.ID*), SPPRs (*S.ID*), mass storages (*M.ID*), and library routines (*L.ID*). All these identification numbers act as processing environment specifiers. The way they are used is indicated in Table 1. The short specification can be used only if the missing specifiers are known from the context.

Table 1. Processing Environment Specifiers

Item	Mnemonic	Full specification	Short specification	Example
CLUSTER	C	C(C ID)	C(C ID)	C(4)
HOST	H	H(H ID)	H(H ID)	H(7)
JOB	J	J(H ID P ID J ID)	J(J ID)	J(7 5 3) or J(3)
LIBRARY ROUTINE	L	L(C ID L ID)	L(L ID)	L(28 4) or L(4)
MASS STORAGE	M	M(C ID)	M(C ID)	M(4)
PROCESS	P	P(H ID P ID)	P(P ID)	P(7 5) or P(5)
SPPR	S	S(C ID S ID)	S(S ID)	S(12 4) or S(4)
TASK	T	T(H ID P ID J ID T ID)	T(T ID)	T(7 5 3 1) or T(1)

The following identification numbers and related pieces of information are needed to allocate and run the task: cluster ID (C.ID), library routine ID (L.ID), program parameters (or their locations), and the data (or their locations). Data for a task reside either in a single memory block of one of the system resources (host or SPPR), or in a number of memory blocks, possibly some in the hosts and others in the SP processors. Each system resource containing a data block keeps a list of all tasks that will need or that might need that data block (until permission is given to delete that data block). So, if a task needs a data block it must know the ID of the system resource currently holding that data block. When requesting the data block, the task has to specify its own ID (T.ID). Each task is associated with a vector ($\bar{D}$), the elements of which define the sources of input data for that task. A scalar E is also associated with each task. Its form is either E = X or E = (C.ID S.ID). It specifies which SPPR executed that task. Initially

the value of E is undefined, i.e., E = X. When a task is assigned to an SPPR, this processor will set the value of E to point to itself, i.e., E = [C ID S ID]. Once a task is executed, the output data will be stored in the local memory of the SPPR that executed that task.

Each host and SPPR has attached to it a task allocation controller. This controller is a hardware device which executes special purpose dedicated software to interface the host or SPPR to the interconnection network and to the rest of the system. The controllers are inserted between the host or SPPR and the network. The controller can be implemented as a VLSI chip and will be, hereafter, referred to as *the LOCO station*. The LOCO station controls the access to all the buses and executes the task allocation procedure. So hosts and SPPRs are free of these activities, which has a number of positive implications on system expandability, reliability, and compatibility of various heterogeneous SPPRs. Different access schemes can be used on the buses. The concept of the carrier sense multiple access with collision detection (CSMA/CD) seems to be the most suitable. However, the analysis of possible access schemes will not be presented as a part of this work. When the station acquires a bus, it broadcasts the message with the destination address in its heading. The message is accepted only by the station that matches the address from the message header. On the data and software buses, the station responds only if it recognizes its own address in the message heading. On the allocation bus, each station is responding to three types of addresses: (a) cluster address (C ID), (b) station address, i.e., SPPR address (C ID, S ID), and (c) the address of the train (to be defined later) currently located at that station (H ID, P ID, J ID). If the address in the message header consists of a C ID only, the message will be accepted by the station associated with the first idle SPPR in the chain of the cluster C ID (this can be ensured by appropriate daisy chaining). If the address in the message header consists of both a C ID and S ID, the message will be accepted by the specified SPPR. If the address in the message header consists of a H ID, P ID, and J ID, the message will be accepted by the SPPR currently in possession of the train (to be defined later).

IV. TASK ALLOCATION PROCEDURE

The basic idea in the LOCO approach to distributed task allocation is to use the processing environment specifiers (see Table 1) in addition among different tasks for the SPPRs they need. The specifiers define which job is competing for which SPPR type (indicated by the S ID). Once the SPPR is assigned to a job, in order to execute one of its tasks, the SPPR is loaded with the necessary library routine, parameters and data, and the execution of the task can start. When the execution of the task is completed, the job will compete for the next SPPR that it needs, and so on.

The LOCO procedure will be first presented through an example and then it will be generalized. Assume that the host H(H ID) = H(7) is running a process P(H ID, P ID) = P(7,5) or abbreviated P(5), with a job J(H ID, P ID, J ID) = J(7,5,3) or abbreviated J(3). Assume that J(3) consists of four tasks interrelated as in Fig. 1 and abbreviated as T(1), T(2), T(3), and T(4). Assume that task T(1) has to be executed in cluster C(2) under the control of the library routine L(5), T(2) in C(4) under L(15), T(3) in C(3) under L(13), and T(4) in C(1) under L(13). It follows that data for task T(1) reside in host H(7), for T(2) in H(7), for T(3) in the SPPR that executed T(1), and for T(4) in the SPPRs that executed T(2) and T(3). The final data produced by job J(3) reside in the SPPR that executed T(3) and T(4).

One possible interpretation of the above specified job may be as follows. T(1) refers to the transformation of a flying object image and sends the appropriate SPPR of the SIMD type. T(2) refers to processing of the seismic signal (that may contain some information corresponding to the object launching) and is best executed on a specialized pipelined array processor. T(3) refers to image understanding and requires the appropriate SPPR of the MIMD type. Output data from this task are for some reason needed by the job source. Finally, T(4) refers to intelligent retrieval from a knowledge base within an expert system which is oriented to identification of flying objects. The expert system needs input data from T(2), T(3), and the job source. Its output data are needed in the job destination (same or another host), and also in the job source, e.g., for updating of relevant information. Another possible interpretation may be in the domain of the medical experiment, where T(1) and T(3) refer to processing and understanding of the scanner image, T(2) to processing of an EEG signal, and T(4) to intelligent retrieval from appropriate knowledge base within an expert system for medical diagnosis.

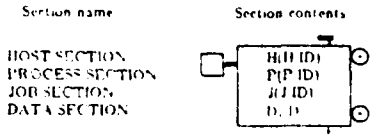


Figure 4 Structure of the Locomotive and Contents of Different Locomotive Sections.

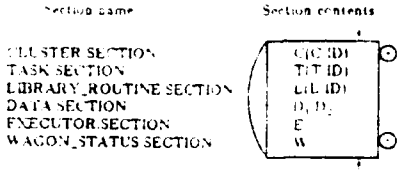


Figure 5 Structure of the Wagon and Contents of Different Wagon Sections. Depending on the LOCO version, the DATA SECTION may contain either data or pointers to data.

Now we describe the way in which the LOCO procedure will treat the above specified job. Once the job J(3) is defined in the process P(5), the host's station corresponding to H(7) will create the message (train) that consists of a number of submessages. One of them is dedicated to the job J(3) as a whole *(the LOCOmotive)*. The others are dedicated to different tasks *(the wagons)*.

The structure of the locomotive is shown in Fig. 4. It consists of four sections. Sections HOST.SECTION, PROCESS.SECTION, and JOB.SECTION define the processing environment of the corresponding job. Section DATA.SECTION defines the tasks that produce the final data needed by this job (other tasks are producing the intermediate data only). Since the locomotive is playing the vital role in the task allocation approach under consideration here, it is referred to as the LOCO approach. Note that the above description of the locomotive implies the case when the job source and the job destination are the same. For the case when job source may be different than the job destination, only a minor modification of the locomotive is required.

The structure of the wagon is shown in Fig. 5. Each wagon W(k), k = 1,2,..., consists of six sections. Section CLUSTER.SECTION specifies the cluster in which the task corresponding to that wagon has to be executed. Section TASK.SECTION specifies the task corresponding to the wagon. Since the wagon is always appended to the locomotive, the short specification of the task can be used. The full specification can be obtained by combining this section and the first three sections of its locomotive. The LIBRARY_ROUTINE.SECTION specifies the library routine to be used in the task execution. Typically, this section will also contain the parameters to be passed to the routine, or at least, the pointers to these parameters. The DATA.SECTION specifies the tasks that produce data for the task corresponding to the wagon. The EXECUTOR.SECTION specifies the particular SPPR that executed the task corresponding to that wagon. Before the execution of the job starts, it is not known which SPPR will do the execution of which task. So, as indicated earlier, the contents of this section is initially E=X, as mentioned earlier. The WAGON_STATUS.SECTION contains the specifier W that indicates if the task corresponding to this wagon is currently under execution somewhere in the system (W=IMAG) or not (W=REAL). If W = REAL and the wagon is behind the locomotive, its execution is completed. If W = REAL and wagon is in the front of the locomotive, its execution did not start yet.

For the particular case of Fig. 1, the initial form of the train is given in Fig. 6a. Initially, the locomotive is pushing the train. The front wagon corresponds to T(1), the next one to T(2), etc. Of course, the appropriate preamble should be appended to the front, and the appropriate cyclic redundancy check (CRC) for error detection purposes to the end of the train. Once the train is created, the host's station will compete for the allocation bus and after accessing it, the station will broadcast the train. The CLUSTER.SECTION of the front wagon (now W(1)) has the function of the train destination address, and the train will end up in the first currently idle SPPR of the cluster C(2). When the train is accepted an acknowledgement will be sent to the train transmitter. The ID of the train transmitter can be found by

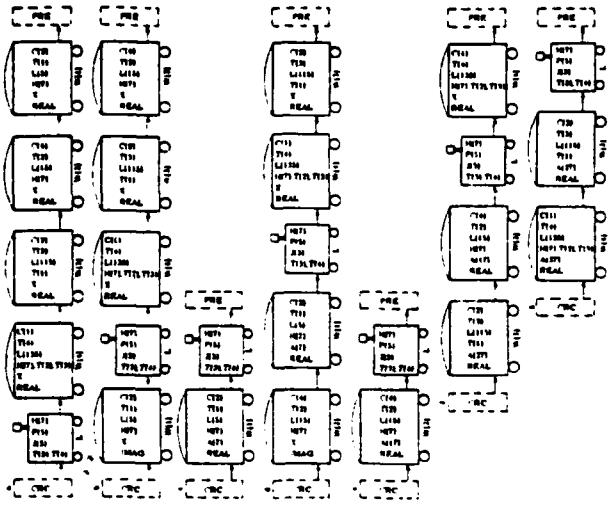


Figure 6. Different Forms of the Train During Execution of the Job from Figure 1.

PRE – Preamble CRC – Cyclic redundancy check

examining the contents of the locomotive. If the first SPPR in the chain is currently busy, it will pass the train to the next SPPR in the chain. If none is available, the train will "wait" in the queue until the first SPPR becomes available. This "queue" may physically exist in the form of the closed cluster-dedicated loop within which the train is propagated until allocated. This is indicated by dashed lines in Fig. 2c. The station that transmitted the train will wait for the acknowledgement. When the acknowledgement is received it will clean up the buffer in which the train was stored and will use it for the other purposes. Here an error-free channel is assumed.

Assume one of the SPPRs is free (e.g. S(7)). Using the allocation bus, the SPPR S(7) will acknowledge the receipt of the train and at the same time it will request data from all the sources specified in the DATA.SECTION of W(1), i.e., from P(7,5). Using the software bus, the SPPR S(7) will request the library routine specified in the program section of W(1), i.e., L(5). In the meantime, before the data and program arrive, the SPPR S(7)'s station will examine W(2) to see if T(2) can run concurrently with T(1). This is indicated by the contents of the DATA.SECTION of W(2). In this example, concurrency is possible. Note that the wagon must be in the station while the SPPR works on its load. It will have W=IMAG during that time. So, W(1) will be removed and the imaginary copy of W(1) will be appended to the back of the train. A copy of the locomotive will be saved at the station along with the wagon. The train (see Fig. 6b) will now be broadcast and hopefully accepted by one of the stations in C(4), e.g., S(17). The station S(17) will acknowledge the receipt of the train, will request its data and program and will examine W(3) for possible concurrency. This time concurrency will not be possible, since T(3) needs data from T(1) and the wagon corresponding to T(1) is imaginary which means that T(1) is not yet completed. So, the train will sit in the station S(17) for some time. So far, our example clearly points to the ability of the LOCO procedure to exploit maximally the existing parallelism on the task level. Other more sophisticated forms of parallelism could be handled by the LOCO procedure equally well.

After some time, T(1) will be completed. The station S(7) will place the output data into its local memory and will "remember" that the data will be needed by T(7,5,3,3). That information is obtained from the train while it is at the station. The station S(7) will set up E=S(7) in the wagon W(1), will append W(1) to the train, and will broadcast it (see Fig. 6c).

The message from Fig. 6c will be accepted by the station which is currently in the position of the train, i.e., S(17). The station S(17) will now exchange the imaginary wagon with the real one, and will reexamine if T(3) can run concurrently. Since now it can, the train of the form indicated in Fig. 6d will be broadcast. Assume that this train will be accepted by S(2,27) and that T(4) will be executed in S(1,37). In that case, the train will have the forms indicated in Figs. 6e, 6f, and 6g. Note that a wagon will be destroyed when it is not needed any more. Finally, the locomotive is pulling the wagons. Once the train from Fig. 6g is accepted by P(7,5) it will request the final data from S(2,27) and S(1,37). At last, P(7,5) will broadcast the permission to delete all memory blocks corresponding to J(7,5,3)

Our example described the basic idea of the LOCO approach. A more rigorous definition can be easily derived from this example. However, note that the LOCO approach is more powerful than indicated by the example. Instead of the topology from Fig. 1, any topology can be used. Next, in the example used here, the schedule of the train and its load (i.e., which type of SPPRs will be visited and what will be the data sources) is set up at the time when the train was created. However, each task can be given the possibility to change the contents of all the wagons corresponding to the tasks not yet executed. In that case the task execution is made conditional, as well as the data to be used. Also, as indicated earlier, each SPPR can be given the possibility to treat each accepted task as a secondary process which can generate secondary jobs and secondary tasks, where a new secondary train has to be associated with each secondary job. Also, it is very important to note that, under assumption #5, *all possible parallelism on the task execution level can be fully exploited by the LOCO procedure.* The actual extent to which the parallelism will be exploited depends upon how the train is composed when it is generated, i.e., the way in which the job is decomposed into tasks and the way in which the wagons are ordered.

Note that the LOCO procedure can exist in various versions. In one version, the train first competes for the appropriate SPPR and then collects the input data (specified by the pointers in the train). In a variation, the train first collects the data needed for the task and then competes for the appropriate SPPR. The former version was explained in the example, since we feel it is simpler. It needs a smaller queueing buffer in each cluster, but is less time-efficient. The latter version will be treated in the performance analysis to follow. It needs a larger queueing buffer in each cluster, but is more time-efficient.

V. MODELLING

We first develop a model of complex multitask job (intertask model) which is applicable to both the LOCO and LB approaches. Then we develop the models of the task execution time (intratask model), separately for the LOCO and LB approaches. Load balancing has attracted a lot of research interest, and some very good work has been reported recently [e.g., ChoAb82, NiHwa81, TanTo84, WahJu83] However, here under the term LB approach we consider the approach which is obtained by applying the principles of load balancing to the system architecture of Fig. 2b.

A. Model of the Multitask Job (Intertask Model)

We assume a complex multitask job that consists of J tasks (running serially and/or concurrently). Each task is serviced by a generalized service station (*GSS*). Activities of the GSS include allocation of the task to one of the appropriate SPPRs, collection of input data from appropriate data sources, collection of the library routine from the appropriate mass storage unit, and execution of the task. The only difference between the GSS for the LOCO and LB approaches is in the allocation of the task to one of the appropriate SPPRs. So the differences are within the GSS and are not visible on the level of the intertask model. Consequently, both procedures can be represented by the same open queueing network model [Kobay78], as indicated in Fig. 7.

Our analytical model based on queueing theory incorporates only the most essential parameters of two procedures under consideration. We are forced to such an approach by the inherent limitations of queueing theory.

We assume an infinite population queueing network. Task generation does not depend on the number of tasks currently existing in the network. Task generation is governed by the Poisson process. Routing of tasks abides by a first-order Markovian chain. Queueing discipline at each GSS can be any work-conserving one. Service time is exponentially distributed. The task destination is capable of absorbing all tasks departing from the system. The observation interval is long enough so that the system can reach a steady state. Under these conditions, Jackson's decomposition theorem [Kobay78] holds, and the steady-state distribution of the probability that the network is in state $\bar{n}$ is given by:

$$P(\bar{n}) = \prod_{i=1}^{N_q} p_i(n_i)$$

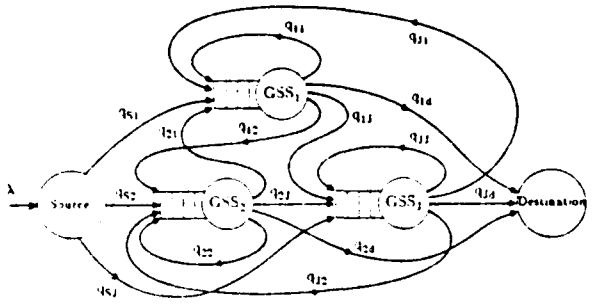


Figure 7 Open Queueing Network Model of a Multitask Job for the LOCO and LB Approaches.

q_{ij} – Branching probabilities for different tasks within the job
G_{SS} – Generalized service station
J – Number of tasks in the job
i,j = S(Source), 1,2,....,J,d(Destination).
λ – Poisson arrival rate at the source node

where $p_i(n_i)$ is the marginal distribution of the variable n_i (i=1,....,N_S), and N_S refers to the number of possible states [Kobay78]. Elements of the vector $\bar{n}$ refer to the number of tasks in each of N_S SPPRs of a given cluster. This conclusion implies that execution of different tasks within a complex multitask job can be analyzed independently one from another, regardless of the intertask data dependency and other relevant parameters. The same holds for both the LOCO and LB approaches. On the basis of this conclusion, in the next subsection the intratask models for the LOCO and LB approaches are introduced and used later for their comparative performance analysis.

B. Model of the Task Execution (Intratask Model)

We consider a task which belongs to a complex (multitask) job, and its execution in the GSS. In general, input data for the task reside in one or more of the hosts or SPPRs. Now we assume that input data reside in a given host. Also, we assume that when the task is ready for execution before its allocation, first the input data have to be requested from the host. The same applies for both the LOCO and LB approaches. The model of the host as a source station for data retrieval is given in Fig. 8a. The FIFO queueing discipline is assumed. If the data request at the host i (i=1,....,N_H) is a Poisson process characterized by the arrival rate $\lambda_{H,i}$, the probability density function (*p.d.f.*) of the data request interarrival intervals is given by:

$$f_A^{H,i}(t) = \lambda_{H,i} e^{-\lambda_{H,i} t} \quad ; \quad t \geq 0$$

The value of $\lambda_{H,i}$ can easily be measured as:

$$\lambda_{H,i} = \lim_{t \to \infty} \frac{\text{number of data requests}}{t}$$

If the service time (i.e., data retrieval) at the host i (i=1,....,N_H) is an exponential process characterized by the service rate $\mu_{H,i}$, then the p.d.f. of the service time is given by:

$$f_s^{H,i}(t) = \mu_{H,i} e^{-\mu_{H,i} t} \quad ; \quad t \geq 0$$

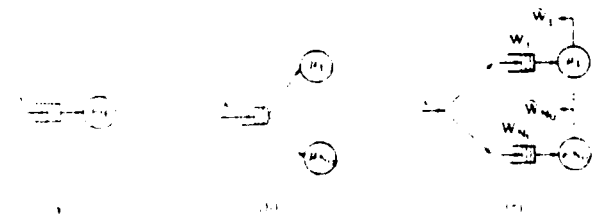


Figure 8 Elements of the Task Execution Model.

(a) The host as a service station for data retrieval
(b) The LOCO approach
(c) The LB approach
W_i (i=1,....,N_C) – Estimated waiting time
$\hat{W}_i$ (i=1,....,N_C) – Real waiting time

The value of $\mu_{H,i}$ can easily be measured as:

$$\mu_{H,i} = \frac{\bar{C}_{H,i}}{\bar{D}_{H,i}}$$

where $\bar{D}_{H,i}$ refers to the average number of instructions executed during data retrieval, and $\bar{C}_{H,i}$ to the average number of instructions executed in a unit of time. The utilization factor is given by:

$$\rho_{H,i} = \lambda_{H,i} / \mu_{H,i}$$

The waiting time p.d.f. at the host is given by:

$$f_w^{H,i}(t) = \mu_{H,i} \rho_{H,i} (1-\rho_{H,i}) e^{-\mu_{H,i}(1-\rho_{H,i})t} \quad ; \quad t \geq 0$$

Finally, the response time p.d.f. at the host is given by:

$$f_T^{H,i}(t) = f_s^{H,i}(t) \odot f_w^{H,i}(t) = \mu_{H,i}(1-\rho_{H,i}) e^{-\mu_{H,i}(1-\rho_{H,i})t} \quad ; \quad t \geq 0$$

where $\odot$ stands for convolution. The average response time for data retrieval at the host is given by:

$$\bar{T}^{H,i} = \int_{-\infty}^{\infty} t \cdot f_T^{H,i}(t) dt$$

Next, after the data are requested and obtained, the task is allocated to a SPPR according to the existing task allocation procedure. Note that our model of the LOCO procedure concentrates on a single cluster. So the parallel execution of different tasks in different clusters is incorporated only indirectly.

In the case of the LOCO approach, the task is sent to the queue corresponding to the appropriate logical cluster. As indicated earlier, this queue may physically exist in the form of the closed cluster-dedicated loop within which the train is propagated until allocated (see Fig. 2c). So the logical cluster can be modelled as a single multiple server service station. The FIFO queueing discipline is assumed. We assume a Poisson arrival of the tasks (due to the decomposition of the complex multitask jobs with Poisson arrivals) with the arrival rate at cluster j (j=1,....,N_C) equal to $\lambda_{c,j}$. We assume an exponential service at each SPPR in the cluster, with the service rate equal to $\mu_{c,j}$. Thus, an M/M/m queueing system is assumed, where m = N_U. Both $\lambda_{c,j}$ and $\mu_{c,j}$ can easily be measured in real systems. The traffic intensity of the cluster is given by $a_{c,j} = \lambda_{c,j}/\mu_{c,j}$ and the utilization factor by $\rho_{c,j} = \lambda_{c,j}/(m_j \cdot \mu_{c,j}) = a_{c,j}/m_j$, where m_j is the number of SPPRs in the cluster j. Note that $m_j = N_U$ under the assumption that each cluster contains the same number of SPPRs. Now we are temporarily removing that assumption in order to make the results more general. The response time p.d.f. at the cluster j is given by convolution of the appropriate service time p.d.f. and waiting time p.d.f.:

$$f_T^{c,j}(t) = f_s^{c,j}(t) \odot f_w^{c,j}(t)$$

In expanded form this reads:

$$f_T^{c,j}(t) = \begin{cases} m_j \mu_{c,j}(1-\rho_{c,j}) E_2(m_j a_{c,j}) \dfrac{e^{-m_j \rho_{c,j}(1-\rho_{c,j})t} - e^{-\mu_{c,j} t}}{1 - m_j(1-\rho_{c,j})}, & t \geq 0;\ \rho_{c,j} > \dfrac{m_j - 1}{m_j} \\ m_j \mu_{c,j}(1-\rho_{c,j}) E_2(m_j a_{c,j}) \dfrac{e^{-\mu_{c,j} t}}{m_j(1-\rho_{c,j}) - 1}, & t \geq 0;\ \rho_{c,j} < \dfrac{m_j - 1}{m_j} \end{cases}$$

where

$$E_2(m_j a_{c,j}) = \left[\frac{a_{c,j}^{m_j}}{m_j!(1-\rho_{c,j})} \right] \Big/ \left[\sum_{k=0}^{m_j - 1} \frac{a_{c,j}^k}{k!} + \frac{a_{c,j}^{m_j}}{m_j!(1-\rho_{c,j})} \right]$$

Finally, the average response time for task execution in the logical cluster is given by

$$\bar{T}^{c,j}(\text{LOCO}) = \int_{-\infty}^{\infty} t \cdot f_T^{c,j}(t) dt$$

Note that $\bar{T}^{c,j}$(LOCO) refers to the average time that a task spends in the LOCO cluster after the data are retrieved.

As already mentioned, we consider a task which is part of a complex multitask job. So the model has to incorporate both the response *time for data retrieval and response time for task execution*. Passing the output data from the task under consideration, to the following task is incorporated into the model of the following task.* In conclu-

*Note that final data have to be forwarded to the job destination. However this can be neglected if the number of serially executed tasks is large enough.

sion, the GSS in the case of the LOCO approach can be modeled as a cascade of the models from Figs. 8a and 8b. This issue will be addressed in Section VI.

In the case of the LB approach, the source of the task first inquires about the load of different SPPRs appropriate for that task. After that information is obtained, the data for the task are requested and obtained and the task is sent to the queue of the SPPR which reported the minimal load (in terms of the total estimated execution time of all tasks currently waiting in its queue). Note that the reported load represents the estimated value (W), and not the real value (W). The minimal reported value is not necessarily the absolute minimal value. This is indicated in Fig. 8c. In conclusion, the GSS for the case of the LB approach can be modelled as a cascade of the models from Figs. 8a and 8c. Under the same conditions as in the case of the LOCO approach, if the load estimation is ideal, the average task response time at the SPPR should be given by the same equation for both approaches:

$$\bar{T}_j(LB;IDEAL) = \bar{T}_j(LB;\sigma=0) = \bar{T}^{c-j}(LOCO)$$

where σ refers to the standard deviation of the task execution time estimate. In this case, the LOCO and LB approaches are characterized by the same performance.

For fair comparison of the LOCO and LB approaches, a model for the LB approach has been chosen which maximally favorizes the LB approach. Consequently, a multiserver model has been chosen, with information on the standard deviation of the task execution time estimate incorporated into the service time p.d.f.

According to the Kingman-Kollerstrom approximation [Klein76], the waiting time distribution in a G/G/m system is given by:

$$W(t) = 1 - e^{-\frac{2(m\mu)(1-\rho)}{\sigma_a^2+\sigma_s^2/m^2}t}$$

where σ_a^2 refers to the variance of the interarrival time, and σ_s^2 to the variance of the service time. The waiting time p.d.f. $f_w(t)$ is a derivative of the above given W(t). Using this approach we evaluate $\bar{T}_j(LB)$ for the $M/G/N_U$ system. Note that the LB approach is characterized by: $m_j = N_U = m$, $\mu_{c,j} = \mu$, and $\lambda_{c,j} = \lambda$. For G we select the gamma distribution defined by:

$$f_g(t) = \begin{cases} \frac{\alpha(\alpha t)^{\beta-1}e^{-\alpha t}}{\Gamma(\beta)} & ; \ t \geq 0 \\ 0 & ; \ t < 0 \end{cases}$$

with $\beta = 3$ and $\alpha = A\mu$. We have chosen an integer β to simplify the analysis, without affecting its generality. The value β=3 has been chosen as it is the case when the gamma distribution closely corresponds to the normal distribution [Klein76]. Parameter A has been incorporated to enable more flexible variations of the mean and the variance. For the gamma distribution, the mean and variance are equal to β/α and β/α^2, respectively [Kobay78]. For selected values of α and β, the service time p.d.f. is given by:

$$f_s(t) = \begin{cases} \frac{(A\mu)^3 t^2 e^{-A\mu t}}{2} & ; \ t \geq 0 \\ 0 & ; \ t < 0 \end{cases}$$

with the mean equal to $\frac{3}{A\mu}$, and the variance equal to $\frac{3}{A^2\mu^2}$. Note that $\int_0^{3/\alpha} f_g(t)dt = 0.575$, and for the exponential distribution we have: $\int_0^{1/\alpha} \mu e^{-\mu t}dt = 0.63$. This approach allows us to evaluate the LB system performance for different values of σ (σ was defined earlier), and for various appropriate values of A.

The response time p.d.f. for LB system is given by convolution of appropriate service time p.d.f., and waiting time p.d.f.:

$$f_j^{LB}(t) = f_j^s(t) \otimes f_j^w(t)$$

where j = 1,...,N_U (number of SPPRs in a cluster). In expanded form this reads:

$$f_j^{LB}(t) = \begin{cases} \frac{A^3\mu^3 C}{(A\mu-C)^3}e^{-Ct} & ; C < A\mu \\ \frac{A^3\mu^3 C[t^2(C-A\mu)^2-2t(C-A\mu)+2]}{2(C-A\mu)^3}e^{-A\mu t} & ; C > A\mu \end{cases}$$

where:

$$C = \frac{2A^2 m\mu(1-\rho)}{A^2\rho^2+3}$$

Finally, the average response time for task execution (after the data are retrieved) is given by:

$$\bar{T}_j(LB) = \int_{-\infty}^{\infty} t f_j^{LB}(t)dt$$

Another possibility for dealing with LB approach is by using the following assumption: If the load estimation is nonideal ($\sigma \neq 0$), then the average task execution time for the LB approach should be given by:

$$\bar{T}_j(LB;\sigma\neq 0) = \bar{T}^{c-j}(LOCO) \cdot \Omega(\sigma;m_j)$$

where $\Omega(\sigma;m_j)$ is the modification function for the LB approach, in the case when the data retrieval time is not taken into consideration. The function Ω characterizes the load estimation. The form of function $\Omega(\sigma;m_j)$ depends on the type of estimation. We assume that statistical characteristics of the estimation error $\rho_i = W_i - W_1$ (i=1,...,m_j) at each station are the same and given by the zero-mean Gaussian distribution of the form:

$$\omega(\rho_i) = \frac{1}{\sqrt{2\pi\sigma^2}}e^{-\frac{\rho_i^2}{2\sigma^2}}; \quad -\infty < \rho_i < +\infty$$

As already indicated, σ is equal to the standard deviation of the load estimation. It is very difficult to obtain an analytic form of the function $\Omega(\sigma;m_j)$. The family of curves in Fig. 9 is obtained by simulation. In this figure, the value of σ is treated relatively to the average execution time of all tasks involved in the simulation ($\bar{T}$). The level of detail in our simulation model was chosen to correspond to the level of detail in our analytical model.

Using the method of empirical-functions smoothing and applying it to Fig. 9, it is possible to derive analytical expression for $\Omega(\sigma/\bar{T};m_j)$. We assume that the function Ω could be given by the following analytical formula:

$$\Omega(x) = K \cdot e^{-(x-2)^{m-2}} + 1$$

Coefficient K depends on m_j. It has been determined that it is equal to 0.77, 1.53, 2.24, and 2.83, for m_j equal to 2, 4, 8 and 16, respectively. The standard deviation is less than 2.1% in all cases. Using these results, we get an estimation for coefficient K which is characterized by a standard deviation less than 5%, for all selected cases. This value reads:

$$K = \frac{m_j - 1}{m_j} \cdot \frac{\log_2(m_j)+2}{2}$$

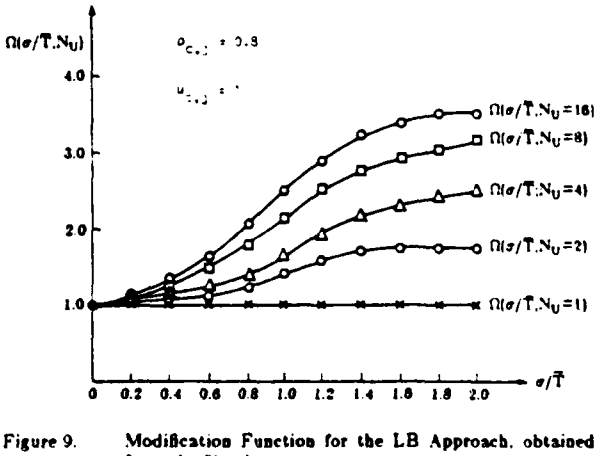


Figure 9. Modification Function for the LB Approach, obtained from the Simulator.

- σ – Standard deviation of the task execution time estimate
- $\bar{T}$ – Average task execution time (execution only, no waiting)
- N_U – Number of units in the cluster

It is possible to use this approximation only in the cases when we have values for the LOCO approach and we need to generate the values for the LB approach, under the conditions of our simulation. Our simulator is of the "self-driven" type [Kobay78], and is implemented in the SLAM language. We have fully followed the methodology of [Kobay78] for simulation model formulation, simulator implementation, design of the simulation experiments, validation of the simulation model, and analysis of the simulation data. *Throughout the simulation, the traffic of individual SPPRs was kept constant.*

VI. PERFORMANCE ANALYSIS

We consider first the model of the LOCO approach developed in the previous section (Figs. 8a and 8b). According to [Kobay78], the average time that a task spends in the system is given by:

$$\bar{T}_{ij}(\text{LOCO}) = \int_0^\infty t\, f_{T(ij)}(t)dt$$

where:

$$f_{T(ij)}(t) = f_T^{\mu_H}(t) \otimes f_T^{\mu_c}(t)$$

We assume that the input data reside in the host i (i=1,....N_H), and that the task is executed in cluster j (j=1,...,N_C). After applying a series of transformations we get:

$\bar{T}_{ij}(\text{LOCO}) =$

$$\frac{m\,\mu_{c_j}(1-\rho_{c_j})E_2(m_j)}{\mu_{H_i}(1-\rho_{H_i})[(1-m(1-\rho_{c_j})][(m_j\mu_{c_j}(1-\rho_{c_j})-\mu_{H_i}(1-\rho_{H_i})]} \cdot \left[\rho_{c_j} > \frac{m-1}{m_j} \right]$$

$$\left[m_j\mu_{c_j}(1-\rho_{c_j})>\mu_{H_i}(1-\rho_{H_i})\right]$$

$$\frac{\mu_{H_i}(1-\rho_{H_i})E_2(m_j)}{-m_j\mu_{c_j}(1-\rho_{c_j})[(1-m_j(1-\rho_{c_j})][(m_j\mu_{c_j}(1-\rho_{c_j})-\mu_{H_i}(1-\rho_{H_i})]} \cdot \left[\rho_{c_j} > \frac{m-1}{m_j} \right]$$

$$\left[m_j\mu_{c_j}(1-\rho_{c_j})<\mu_{H_i}(1-\rho_{H_i})\right]$$

$$\frac{m\,\mu_{c_j}(1-\rho_{c_j})E_2(m_j)}{\mu_{H_i}(1-\rho_{H_i})[m(1-\rho_{c_j})-1][\mu_{c_j}-\mu_{H_i}(1-\rho_{H_i})]} \cdot \left[\rho_{c_j} < \frac{m-1}{m_j} \right]$$

$$\left[\mu_{c_j}>\mu_{H_i}(1-\rho_{H_i})\right]$$

$$\frac{\mu_{H_i}(1-\rho_{H_i})(m(1-\rho_{c_j})E_2(m_j)}{-\mu_{c_j}[m_j(1-\rho_{c_j})-1][\mu_{c_j}-\mu_{H_i}(1-\rho_{H_i})]} \cdot \left[\rho_{c_j} < \frac{m-1}{m_j} \right]$$

$$\left[\mu_{c_j}<\mu_{H_i}(1-\rho_{H_i})\right]$$

where:

$$a_{c_j} = \frac{\lambda_{c_j}}{\mu_{c_j}} \quad \text{or} \quad \rho_{c_j} = \frac{\lambda_{c_j}}{m_j\mu_{c_j}}$$

and $E_2(m_j)$ was defined earlier in the text. The first two formulas apply to the case $m_j = 1$. The dependence of the $\log_{10}|\bar{T}_{ij}(\text{LOCO})|$ on m_j is presented in Fig. 10 for the case when $m_j=N_{Cj}$ and for different values of μ_{Hi} and ρ_{Hi}. *Note that the total traffic in Fig. 10 is kept constant, regardless of the value of* N_{Cj}. *Consequently, when* N_{Cj} *increases, the individual traffic of each SPPR decreases.* Variance of the total time that a task spends in the LOCO system is given by

$$\sigma^2_{ij}(\text{LOCO}) = \int_0^\infty t^2 f_{T(ij)}(t)dt - \bar{T}^2_{ij}(\text{LOCO})$$

After a series of transformations we get:

$\sigma^2_{ij}(\text{LOCO}) =$

$$\frac{m\mu_{c_j}(1-\rho_{c_j})E_2(m_j)\{2[(1-m(1-\rho_{c_j})][m\mu_{c_j}(1-\rho_{c_j})-\mu_{H_i}(1-\rho_{H_i})]-m\,\mu_{c_j}(1-\rho_{c_j})E_2(m_j)\}}{\mu^2_{H_i}(1-\rho_{H_i})^2[1-m(1-\rho_{c_j})]^2[m\mu_{c_j}(1-\rho_{c_j})-\mu_{H_i}(1-\rho_{H_i})]^2}$$

$$(\rho_{c_j} > \frac{m-1}{m_j} \cdot m\mu_{c_j}(1-\rho_{c_j}) > \mu_{H_i}(1-\rho_{H_i}))$$

$$\frac{\mu_{H_i}(1-\rho_{H_i})E_2(m_j)\{2[(1-m_j(1-\rho_{c_j})][\mu_{H_i}(1-\rho_{H_i})-m_jC_{c_j}(1-\rho_{c_j})]-\mu_{H_i}(1-\rho_{H_i})E_2(m_j)\}}{m_j\mu^2_{c_j}(1-\rho_{c_j})^2[1-m_j(1-\rho_{c_j})]^2[\mu_{H_i}(1-\rho_{H_i})-m_jC_{c_j}(1-\rho_{c_j})]^2}$$

$$(\rho_{c_j} > \frac{m-1}{m_j} \cdot m_j\mu_{c_j}(1-\rho_{c_j}) < \mu_{H_i}(1-\rho_{H_i}))$$

$$\frac{m\mu_{c_j}(1-\rho_{c_j})E_2(m_j)\{2[m(1-\rho_{c_j})-1][\mu_{H_i}-\mu_{H_i}(1-\rho_{H_i})]-m\,\mu_{c_j}(1-\rho_{c_j})E_2(m_j)\}}{\mu^2_{H_i}(1-\rho_{H_i})^2[m(1-\rho_{c_j})-1]^2[\mu_{c_j}-\mu_{H_i}(1-\rho_{H_i})]^2}$$

$$(\rho_{c_j} < \frac{m-1}{m_j} \cdot \mu_{c_j} > \mu_{H_i}(1-\rho_{H_i}))$$

$$\frac{m\,\mu_{H_i}(1-\rho_{H_i})(1-\rho_{c_j})E_2(m_j)\{2[m(1-\rho_{c_j})-1][\mu_{H_i}(1-\rho_{H_i})-\mu_{c_j}]-m\,\mu_{H_i}(1-\rho_{H_i})(1-\rho_{c_j})E_2(m_j)\}}{\mu^2_{c_j}[m(1-\rho_{c_j})-1]^2[\mu_{H_i}(1-\rho_{H_i})-\mu_{c_j}]^2}$$

$$(\rho_{c_j} < \frac{m-1}{m_j} \cdot \mu_{c_j} < \mu_{H_i}(1-\rho_{H_i}))$$

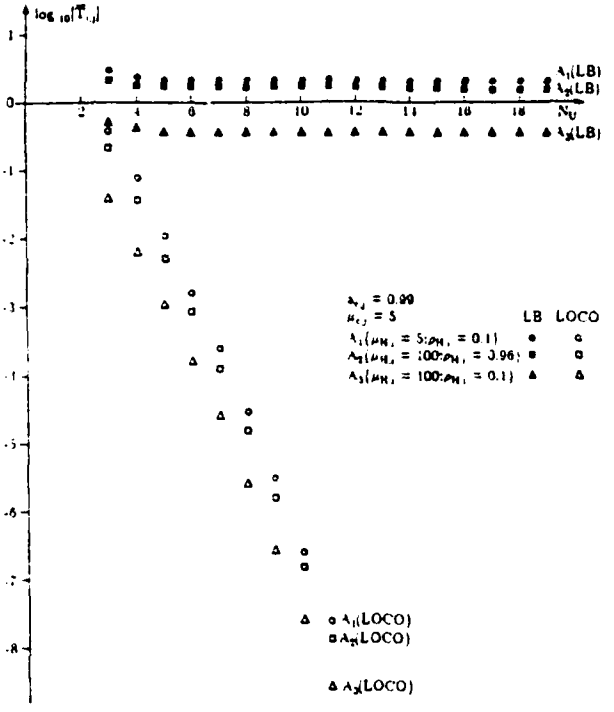


Figure 10. Average Time that Task Spends in the LOCO and LB Systems.

N_U – Number of units in the cluster

Average total time spent in the system, averaged over all possible data sources and task types, is given by:

$$\bar{T}(\text{LOCO/SYSTEM}) = \sum_{i=1}^{N_H}\sum_{j=1}^{N_C} \bar{T}_{ij}(\text{LOCO})p_{ij}$$

where p_{ij} refers to the probability that input data reside in the host i and the task is executed in cluster j. Average queue length of the cluster j (j=1,....N_C) in terms of the number of tasks waiting in the queue associated to cluster j is given by [Kobay78]:

$$\bar{Q}_j = \sum_{n=m_j}^{\infty}(n-m_j)p_n = \frac{a_{c_j}^{m_j}}{m_j!}\frac{\rho_{c_j}}{(1-\rho_{c_j})^2}\cdot p_0$$

where:

$$p_0 = \left[\sum_{n=0}^{m_j-1}\frac{a_{c_j}^n}{n!} + \frac{a_{c_j}^{m_j}}{m_j!}\frac{1}{1-\rho_{c_j}}\right]^{-1}$$

The variance of the queue length is given by:

$$\sigma^2_{Q_j} = \sum_{n=m_j}^{\infty}(n-m_j)^2p_n - \bar{Q}_j^2 = \frac{a_{c_j}^{m_j}}{m_j!}\cdot\frac{1-\rho_{c_j}+2\rho_{c_j}^2}{(1-\rho_{c_j})^3}\cdot p_0$$

where p_0 is defined above, and p_n is the probability of having n tasks in the cluster.

We consider now the model of the LB approach developed in the previous section (Figs. 8a and 8c). According to queueing theory, the average time that a task spends in the system is given by:

$$\bar{T}_{ij}(\text{LB}) = \int_0^\infty t\, f_{T(ij)}(t)dt$$

where:

$$f_{T(ij)}(t) = f_T^{\mu_H}(t) \otimes f_T^{\mu_c}(t)$$

We assume that the input data reside in the host i ($i=1,\ldots,N_H$), and the task is executed in the SPPR of the type j ($j=1,\ldots,N_U$). After applying a series of transformations we get:

$$\bar{T}_{ij}(LB) = \begin{cases} \dfrac{\lambda D A^2}{(A\mu-C)^2(D-C)-C} & ; C < A\mu \\ \dfrac{\lambda \bar{T}_{ij}^2}{(C-A\mu)(D-C)D} & ; D < C \\ \dfrac{A^2\bar{T}_{ij}^2[C-\lambda\mu]^2+(C-\lambda\mu)(D-\lambda\mu)+(D-\lambda\mu)^2]}{(C-A\mu)^2(D-\lambda\mu)+A\mu^2D} & ; C > A\mu \\ \dfrac{CD[2AC-A\mu)^2D-\lambda\mu]^2-2A\mu(C-2A\mu+D)C-\lambda\mu)D-\lambda\mu)^2+A^2\mu^2D-\lambda\mu)^2+(C-\lambda\mu)D-\lambda\mu)^2+(C-\lambda\mu)^2]}{(C-A\mu)^2D-A\mu)^2+A^2\mu^2D} & ; D > A\mu \end{cases}$$

where:

$$C = \frac{2A^2m\mu(1-\rho)}{A^2\rho^2+3} \quad \text{and} \quad D = \mu_{H,i}(1-\rho_{H,i}) .$$

The dependence of the $\log_{10}[\bar{T}_{ij}(LB)]$ on m_j is presented in Fig. 10 for various values of $\mu_{H,i}$ and $\rho_{H,i}$. The plotting is provided for A = 1.75. Note that total traffic in Fig. 10 is kept constant, regardless of the value of N_U. This is the same as in the case of Fig. 10, but different compared to Fig. 9. When N_U increases, the individual traffic of each SPPR decreases, but slower (Fig. 10).

Variance of the total time that a task spends in the LB system is given by:

$$\sigma_{\bar{T}}^2(LB) = \int_0^\infty t^2 f_{\bar{T}_{ij}}(t)dt - \bar{T}_{ij}^2(LB)$$

After a series of transformations we get:

$$\sigma_{\bar{T}}^2(LB) = \begin{cases} \dfrac{A^2\mu^2[2AC-A\mu)^2C-D]-A^2\mu^2D]}{C^2(C-A\mu)^2(D-C)^2} & C < A\mu \\ & \rho_{H,i}(1-\rho_{H,i}) > C \\ \dfrac{A^2\mu^2[(C-A\mu)^2D-C]-C]}{(C-A\mu)^2(D-C)^2D^2} & C < A\mu \\ & \rho_{H,i}(1-\rho_{H,i}) < C \\ \dfrac{A^2\mu^2[(C-A\mu)^2+(C-A\mu)(D-A\mu)+(D-A\mu)^2][(C-A\mu)^2D-A\mu)^2+C]DA^2\mu^2[(C-A\mu)^2+(C-A\mu)(D-A\mu)+(D-A\mu)^2]}{(C-A\mu)^2+(A\mu-D)^2+D^2} & C > A\mu \\ & \rho_{H,i}(1-\rho_{H,i}) < A_\mu \\ \dfrac{CD}{(C-A\mu)^2D-A\mu)^2+A^2\mu^2}\Big\{[2C-A\mu)^2D-A\mu)^2][2C-A\mu)^2D-A\mu)^2-[2C-A\mu)(D-A\mu)][2A\mu+D)A\mu & C > A\mu \\ +[(D-A\mu)^2+(C-A\mu)(D-A\mu)+(C-A\mu)^2]A^2\mu^2]-CD+[2C-A\mu)^2D-A\mu)^2 & \rho_{H,i}(1-\rho_{H,i}) > A_\mu \\ -[2C+D-2A\mu)][2C-A\mu)(D-A\mu)A\mu+[(D-A\mu)^2+(C-A\mu)(D-A\mu)+(C-A\mu)^2]A^2\mu^2]\Big\} & \end{cases}$$

where $D = \mu_{H,i}(1-\rho_{H,i})$, while C and A were defined earlier.

Average total time spent in the system, averaged over all possible data sources and SPPR types, is given by:

$$\bar{T}(LB/SYSTEM) = \sum_{i=1}^{N_H}\sum_{j=1}^{N_U} \bar{T}_{ij}(LB) \cdot p_{ij}$$

where p_{ij} refers to the probability that input data reside in the host i and the task needs the SPPR of the type j. The formulas for $\bar{T}(LB)$ and $\bar{T}(LOCO)$ match each other very closely for $N_U = 1$. Numerical values differ only in the third decimal digit.

For the LB approach, average queue length $\bar{Q}_j(LB)$ could be evaluated using Little's formula [Klein 76]:

$$\bar{Q}_j(LB) = \lambda(\bar{T}_j(LB) - \frac{m\rho}{\lambda}) = \lambda\bar{T}_j(LB) - m\rho$$

where $\bar{T}_j(LB)$ was defined earlier, and index i is omitted. After a series of transformations, we get:

$$\bar{Q}_j(LB) = \begin{cases} \dfrac{\lambda A^2\mu^3}{(A\mu-C)^2 \cdot C} - m\rho & ; C < A\mu \\ \dfrac{\lambda C[(C-A\mu)^2-(C-A\mu)A\mu+A^2\mu^2]}{(C-A\mu)^3} - m\rho & ; C > A\mu \end{cases}$$

where C and A were defined earlier.

Some conclusions may be derived from Figs. 9 and 10. The higher the value of σ (implies $A < 3$), the larger the performance difference between the LOCO and LB approaches, which is expected. In the environment under consideration, as already mentioned in Section II, the values of σ are relatively large due to the fact that, in the AI environment, the correlation between past values and future values of execution times for the same type of task may be very low. This indicates that, for realistic values of σ, the performance difference between the LOCO and LB approaches can be relatively high. For example, according to our simulation, for $\sigma/\bar{T}=1$ (standard deviation of the estimation is equal to the average task execution time), and $N_U=8$ (case of eight SPPRs in each cluster), the total time that the task spends in the system is 2.6 times shorter for the LOCO approach, compared with the LB approach. Note that our simulator neglects the time needed in the LB approach for the inquiry and processing of the information about the load of different SPPRs.

A number of observations have been derived from our analysis. For example, with the given conditions, the higher is the value of N_U, the larger is the performance difference between the LOCO and LB approaches. However, the step of the performance increase is smaller for the higher values of N_U.

VII. CONCLUSION

In this paper a problem was recognized, one of having a large number of special purpose processing resources (SPPRs) shared by a number of hosts. Processing structures of this type will arise in 1990s around AI and other computationally massive applications. Similar processing structures may arise in the high-end computers of the 5th generation. In such a processing structure, it is of crucial importance to have an efficient procedure for the distributed allocation of different tasks among different SPPRs.

Under the assumptions that affect the above described processing structure, a distributed task allocation procedure was introduced which is efficient in a large range of circumstances. Both the task allocation procedure (LOCO) and the underlying system architecture (AIDA) were presented and analyzed.

One of the most desirable features of this approach is that the task allocation controller (the LOCO station) can easily be implemented in a single VLSI chip. The LOCO station acts as an interface between the SPPR and the tasks to be executed by it. The LOCO station enables SPPRs of different types to be incorporated into a monolithic task allocation scheme.

Acknowledgments

The authors thank Professors E. J. Coyle, C. E. Houstis, and B. W. Wah of Purdue University for their creative suggestions.

References

Batch80
Batcher, K. E., "Design of a Massively Parallel Processor," *IEEE Transactions on Computers*, Vol. C-29, No. 9, September 1980, pp. 836-840.

Brady82
Brady, M., "Computer Vision," *Artificial Intelligence*, Vol. 19, 1982, pp. 7-16.

ChoAb82
Chou, T. C. K., Abraham, J. A., "Load Balancing in Distributed Systems," *IEEE Transactions on Software Engineering*, Vol. SE-8, No. 4, July 1982, pp. 401-412.

ChoKo79
Chow, Y.-C., Kohler, W. H., "Models for Dynamic Load Balancing in a Heterogeneous Multiple Processor System," *IEEE Transaction on Computers*, Vol. C-28, No. 5, May 1979, pp. 354-361.

Davis83
Davis, A. L., "Computer Architecture," *IEEE Spectrum*, Vol. 20, No. 11, November 1983, pp.94-99.

Enslo78
Enslow, P. H., "What is a Distributed Data Processing System," *IEEE Computer*, Vol. 11, No. 1, January 1978, pp. 13-21.

Flynn72
Flynn, M. J., "Some Computer Organizations and Their Effectiveness," *IEEE Transaction on Computers*, Vol. C-21, 1972, pp. 948-960.

GaLak84
Gajski, D. D., Lawrie, D. H., Kuck, D. J., Sameh, A. H., "Cedar," *Proceedings of the IEEE Spring Compcon 84*, San Francisco, CA, February/March 1984, pp. 306-309.

GaPaK82
Gajski, D., Padua, D. A., Kuck, D. J., Kuhn, R. H., "A Second Opinion on Data Flow Machines and Languages," *IEEE Computer*, Vol. 10, No. 2, February, 1982, pp. 58-69.

GoGrK83
Gottlieb, A., Grishman, R., Kruskal, C. P., McAuliffe, Rudolph, L., Snir, M., "The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer," *IEEE Transactions on Computers*, Vol. C-32, No. 2, February 1983, pp. 175-189.

Grosz82
Grosz, B. J., "Natural Language Processing," *Artificial Intelligence*, Vol 19, 1982, pp 131-136.

HaWaL83
Hayes-Roth, F., Waterman, D. A., Lenat, D. B., (editors), *Building Expert System*, Addison-Wesley, 1983.

HwCrC82
Hwang, K., Croft, W., Goble, G., Wah, B., Briggs, F., Simmons, W., Coates, C., "A UNIX-Based Local Computer Network with Load Balancing," *IEEE Computer*, Vol. 10, No. 4, April 1982, pp. 55-66.

JenP184
Jensen, E. D., Pleszkoch, N., "ArchOS: A Physically Dispersed Operating System," *IEEE Distributed Processing Technical Committee Newsletter*, Vol. 6, No. SI-2, June 1984, pp. 15-24.

Klein76
Kleinrock, L., *Queueing Systems* (Vol. I and II), John Wiley and Sons, New York, 1976.

Kobay78
Kobayashi, H., *Modeling and Analysis An Introduction to System Performance Evaluation Methodology*, Addison-Wesley, 1978.

KuSiA84
Kuehn, J. T., Siegel, H. J., Adams, G. B., Tuomenoksa, D. L., "The Use and Design of PASM," in *Image Processing: From Computation to Integration*, Levialdi, S., ed., Academic Press, London, 1984.

Lerner82
Lerner, E., "Understanding Speech Proves Tough Task for Machine," *Computer Design*, September 1982, pp. 41-44.

MaBe81
Maples, N., Beangle, T., "Integrating an Array Processor into a Scientific Computing System," *IEEE Computer*, Vol. 9, No. 9, September 1981, pp 41-44.

McEMa82
McDonough, K. C., Magar, S. S., "A Single Chip Microcomputer Architecture Optimized for Signal Processing," *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, Paris, France, May 1982, pp. 1-5.

MiLe84
Milutinovic, V., Siegel, H. J., "The LOCO Approach to Coordinated Task Allocation in AIDA by VERDI," *Purdue University Technical Report*, TR-EE 84-49, November 1984.

Milu83
Milutinovic, V., "A High-Level Language Architecture Bit-Slice Based Processor and Associated System Software," *Microprocessing and Microprogramming*, Vol. 12, Nos. 3 and 4, October/November 1983, pp 142-151.

MiWa83
Milutinovic, V., Waldschmidt, K., "A High-Level Language Architecture for Time-Critical Dedicated Microprocessing," *Microprocessing and Microprogramming*, Vol. 12, No. 1, August 1983, pp 23-42.

MuKaM83
Murakami, K., Kakuta, T., Myazamu, N., Shibayama, S., Yokota, H., "A Relational Data Base Machine: First Step to Knowledge Base Machine," *Proceedings of the ACM International Symposium on Computer Architecture*, Stockholm, Sweden, June 1983, pp 423-425.

NiHwa81
Ni, L. M., Hwang, K., "Optimal Load Balancing Strategies for Multiple Processor Systems," *Proceeding of the IEEE/ACM International Conference on Parallel Processing*, Bellaire, MI, August 1981, pp 352-357.

PoCoS83
Porter, D. R., Couch, P. R., Schelin, J. W., "A High-Speed Fiber Optic Data Bus for Local Data Communications," *IEEE Journal on Selected Areas in Communications*, Vol. SAC-1, No. 3, April 1983, pp. 479-488.

RaBGo75
Rabiner, L. R., Gold, B., *Theory and Applications of Digital Signal Processing*, Prentice-Hall, 1975.

Rober84
Roberts, D., "A Study of the Artificial Intelligence Software Environment," *Purdue University Internal Report*, EE 496-84, August 1984.

ShDaR82
Shoch, J., Dalal, Y., Redell, D., Crane, R., "Evolution of the Ethernet Local Computer Network," *IEEE Computer*, Vol. 10, No. 8, August 1982, pp. 10-27.

Siege84
Siegel, H. J., *Interconnection Networks for Large-Scale Parallel Processing: Theory and Case Studies*, Lexington Books, Lexington, Ma 1984.

SiSiK81
Siegel, H. J., Siegel, L. J., Kemmerer, F., Mueller, P. T., Smalley, H. E., Smith, S. D., "PASM: A Partitionable SIMD/MIMD System for Image Processing and Pattern Recognition," *IEEE Transactions on Computers*, Vol. C-30, No. 12, December 1981, pp. 791-801.

StoSh82
Stolfo, S. J., Shaw, D. E., "DADO: A Tree-Structured Machine Architecture for Production Systems," *Proceedings of the AAAI National Conference on Artificial Intelligence*, Pittsburgh, PA, 1982.

SuHoS81
Sussman, G. J., Holloway, J., Steel, G. L., Bell, A., "Scheme-79 – Lisp on a Chip," *IEEE Computer*, Vol. 14, No. 7, July 1981, pp. 10-21.

SwFuS77
Swan, R. J., Fuller, S. H., Siewiorek, D. P., "Cm* - A Modular Multi-Microprocessor," *Proceedings of the National Computer Conference*, Dallas, TX, June 1977, pp. 637-644.

TanTo84
Tantawi, A. N., Towsley, D., "Optimal Load Balancing in Distributed Computer Systems," *University of Massachusetts Technical Report*, RC-10346, January 1984.

TreLi82
Treleaven, P. C., Lima, I. G., "Japan's Fifth-Generation Computer Systems," *IEEE Computer*, Vol. 15, No. 8, August 1982, pp. 79-88.

Uchi83
Uchida, S., "Inference Machine: From Sequential to Parallel," *Proceedings of the 4CM International Symposium on* Computer Architecture, Stockholm, Sweden, June 1983, pp. 410-416.

Wah84
Wah, B. W., "Comparative Study of Distributed Resource Sharing on Multiprocessors, *IEEE Transactions on Computers*, Vol. C-33, No. 8, August 1984, pp. 700-711.

WahHi82
Wah, B. W., Hicks, A., "Distributed Scheduling of Resources on Interconnection Networks," *Proceedings of the National Computer Conference*, AFIPS Press, pp. 697-709, 1982.

WahJu83
Wah, B. W., Juang, J.-Y., "An Efficient Protocol for Load Balancing on CSMA/CD Networks," *Proceedings of the IEEE Conference on Local Computer Networks*, Minneapolis, MN, October 1983 pp. 55-61.

WahMa84
Wah, B. W., Ma, Y. W., "MANIP: A Multicomputer Architecture for Solving Combinatorial Extremum Search Problems," *IEEE Transactions on Computers*, Vol. C-33, No. 5, May 1984, pp. 377-390.

Paper 11

MIMD Algorithm Analysis: Low Level Algorithm Descriptions

From the *Proceedings of The 5th International Conference on Distributed Computing Systems*, May 1985; Denver, Colorado.

MIMD ALGORITHM ANALYSIS: LOW LEVEL ALGORITHM DESCRIPTIONS

Kirk D. Smith
Leah H. Jamieson

School of Electrical Engineering
Purdue University
West Lafayette, IN 47907

Abstract

This work identifies salient features of MIMD algorithms. A set of language and machine independent MIMD constructs is proposed. In analysis, algorithms are reduced to an equivalent description composed of these constructs. These constructs are at a low level, thus one can analyze the algorithm performance in relation to several MIMD architectures. At the same time, these constructs are at a high enough level to retain the basic structure of the algorithm. The paper focuses on issues of communication and synchronization. Examples from Ada, CSP, Edison, and Path Pascal are given.

Introduction

This work addresses problems in the analysis of MIMD algorithms. Especially in the area of application-driven or algorithm-driven architecture design, one would like to be able to predict the performance of MIMD algorithms on different MIMD architectures. Since few MIMD machines exist, direct execution is generally not possible. Simulation of MIMD processes at a low level is possible but difficult. The work here is intended to provide an extension to traditional algorithm analysis. Features such as inter-process communication and synchronization, which are critical to the performance of MIMD algorithms, are mapped from high level languages to common, relatively low level representations on which analysis can be performed.

The underlying approach will be to extract a few primitive features of parallel algorithms. These features should be comprehensive enough to cover a wide range of language constructs, while being simple enough to correspond to hardware capabilities. The major areas of interest are communications and synchronization. Models are developed to describe many forms of these operations in a uniform notation. In the following sections, some high level language constructs to express communications and synchronization are surveyed. We then identify basic representations to which the high level constructs can be mapped. These low level representations are close enough to the hardware level to allow analysis of the effects of hardware on the execution characteristics. Just as importantly, the representations encapsulate the meaning of the original algorithm. By developing these simple constructs, analysis is simplified and unified.

MIMD Architecture Models

In MIMD machine designs, two memory organizations are common: the Shared Memory Model and the Private Memory Model.

This material is based on work supported by the U.S. Army Research Office under Contract DAAG29-82-K-0101.

The Shared Memory (or Global Memory) Model consists of a set of N Processing Elements (PEs) with no local memory. These are connected through a network to a global store. Examples of this model include the NYU Ultracomputer[1] and C.mmp[2]. The major advantage of the Shared Memory Model is that all processors can access all of memory. This is important in the analysis of algorithms for this type of system. One of the critical design problems in such a system will be the arbitration network. Much research has been devoted to data storage schemes to improve efficiency of the data accesses[3,4].

The Private Memory Model gives each processing element its own memory. The PEs themselves are connected directly through a network. An example of this is the PASM system[5]. The advantages of the Private Memory Model include fast exclusive memory access for each PE to its own memory. The associated cost is the inability to access all of memory directly. Again, this will appear in later discussions of algorithm analysis. Siegel et al.[5] give a good discussion of the relative benefits of each model.

These two models identify one of the largest single differences between various MIMD architectures. Many designs contain aspects of both models. These models are not, therefore, meant to divide MIMD architectures into two classes. These models merely identify two of the most common approaches. For instance, the Texas Reconfigurable Array Computer (TRAC) combines the two given models by providing both private and shared memory[6,7,8]. For the purposes of the analyses presented here, it is sufficient to show that results are valid for both models, so that mixtures of models will also produce valid results.

These models are fairly simple. Thus, they are not intended to take all aspects of a parallel architecture into account. Yet in their simplicity, they distinguish features of an architecture that have a major bearing on its ability to run parallel algorithms. These models will be useful in mapping language constructs to actions in the hardware.

Implementing Language Constructs in Parallel Architectures

Assumptions

In this section, assumptions made in the subsequent analyses are outlined.

There is a distinction to note between tasks or processes and their relationship to PEs. There are several approaches. Each task can be statically assigned to a PE. For Private Memory, this approach is always reasonable since it takes a significant amount of time to copy a task in and out of a PE's local memory. For Shared Memory, this approach is ideal when the number

of tasks is less than the number of PEs. For Shared Memory, another option is to assign a task dynamically to any available PE. Over a much longer time frame, this is feasible for Private Memory as well. Throughout the following discussion, it is assumed that a process is provided with memory resources and a PE. In Shared Memory, the assigned PE may change over time, but the memory resources are unchangeable by external events. For Private Memory, the memory is associated with each PE, so this is fixed. So, to simplify analysis, it will be assumed that once a memory resource is allocated, it stays with the process until the process releases it. No external action can take away or move this resource.

Another assumption is that all communication to other processes or tasks will be performed through some general mechanism. This mechanism will have to handle the details of transferring data from one process to another using the available facilities. In a Shared Memory system, data is transferred through a global store, so all data transfers are accomplished in the same manner. In a Private Memory system, interprocess communication may imply data is transferred between two physical PEs. This would not necessarily be the case when there is more than one process per PE. For the sake of simplicity, the following discussion will assume that a similar execution penalty is incurred for both cases. Except for a trivial MIMD system (2 PEs), or when traffic patterns are explicitly described, there is a higher probability that a transfer will require accessing a physically distinct PE.

Summarizing, to provide for a more concise analysis, a task or process is assumed to be mapped to a logical PE and logical memory. In most cases, this will correspond to a physical PE and physical memory. Future extensions of this analysis may relax this assumption.

In the following sections, two features of parallel algorithms are investigated. Global variables and communications are methods for processes to communicate among themselves. Concurrency specifications identify code which can be executed concurrently. Synchronization allows processes to execute correct algorithms. These facilities are expressed in many different ways in different languages. The goal is to map the multitude of forms of expressions into a common form for analysis. This form should be close enough to the hardware level to allow analysis of the effect of hardware on the execution characteristics. Just as importantly, this form should encapsulate the meaning of the original algorithm.

Global Variables and Communications

In this section, a number of high level language mechanisms for providing shared memory and interprocess communication are surveyed. Implementations on the two MIMD models are discussed.

Local vs. Global Variables

Local variables are considered to be those variables accessed by only one PE, while global variables are accessed by more than one PE. This does not necessarily imply that a local or global variable is kept in a local or global memory. When speaking of variables, local and global refer to a logical association with processes. Different languages support this in different ways. Ada[9,10] assumes data is local to **tasks**. As a consequence of the visibility rules, an object declared in a parent task is visible by the children, thus more than one task can access a variable. In some situations, this is a "dangerous" form of global variables since care must be taken to avoid conflict by one task reading the variable and another task changing it simultaneously. Ada supports a library of implementation-dependent routines which it calls the STANDARD package. To update these globally visible variables in a reliable way, one must use the SHARED_VARIABLE_UPDATE generic procedure defined in the STANDARD package. This is a procedure which insures data integrity in a multiprocessing environment. To do this, it may have to perform some sort of software interlocking. Alternatively, global variables can be implemented as local variables in a special task. This task does nothing except repeatedly accept requests to access the data. That "data manager" task must then **accept** the **rendesvous** by the processes requesting the "global" data.

Similarly, Concurrent Pascal[11,12] uses **monitors** to access global variables. Alternatively, Modula[13,14,15] assumes all variables declared within the main program block are global variables. All other variables are local.

These languages all explicitly provide for global variables. The variables may be accessed in a limited scope or through a special mechanism, but they are readily available.

Interprocess Communication

There are some languages that do not provide for global variables explicitly. An example of this is CSP[16]. This language is a so-called "message based" language. Omitting the global variable construct forces the programmer to use other methods to transfer information between processes. CSP requires all shared information to pass between processes on clearly defined **channels**. This performs a similar function to the traditional shared variable. In fact, through the use of a semaphore and a global variable, a similar operation could be performed in a language that does not support message passing.

Global Variables vs. Communication

Global variables, by definition, contain information relevant to more than one process. This may be in the form of a global variable name, a **monitor,** or a **channel**. Thus, inter-PE communication can be interpreted as a form of global storage. Conversely, global storage can be interpreted as a form of inter-PE communication. Thus, global variables and communications can be used for similar purposes and can be implemented in the same way. In analyzing parallel algorithms, this equivalence can be used to unify many language constructs into a common analysis framework.

Implementation Examples

To illustrate this further, consider implementing an Ada compiler for a Shared Memory system. In order to execute a **SHARED_VARIABLE_UPDATE** the code must access memory set aside for general use. This area of memory is designated as a global storage area and all PEs access it whenever necessary. The hardware accounts for the arbitration and insures that a memory operation by one PE cannot be interrupted by another PE. If this memory is accessed often by several processes, the arbitration can introduce a significant delay. Overall, this scheme is a quite direct interpretation of the language construct.

Now consider implementing the same compiler for a Private Memory system. There is no memory accessible by all PEs. In this case the inter-PE communications network plays a key role. Options include designating a

spare PE as a global memory handler, spreading the global memory among the PEs randomly, or keeping the global memory with the (parent) task where it is declared. When a PE needs to access a memory location, it must pass a short message to the appropriate PE describing the memory operation. The remote PE may return a value or perhaps a write confirmation. This is a somewhat more complicated issue than merely using a hardware arbitration scheme as in Shared Memory.

Conversely, suppose the task at hand is to develop a CSP compiler for a Private Memory machine. CSP provides simple mechanisms for the passing of information on defined channels. There is a natural mapping from the use of channel specifications to the use of an interconnection network.

On the other hand, to provide for channels on a Shared Memory machine, one would have to set aside areas of memory to simulate the hardware channels. For an algorithm with heavy inter-PE communications, it is important that the compiler place these memory areas in a fashion that produces few memory access conflicts. The memory conflict problem is an area of research in itself[3,4], so it is sufficient to note here that this could be a significant problem.

Language vs. Algorithm

In the case of "conventional" languages, it appears that the "natural" mapping of global variables is to a Shared Memory machine. Likewise, CSP and other "message based" parallel languages "naturally" map to a Private Memory machine. It is proposed that this so-called "natural" mapping is actually artificial. In analysis of parallel algorithms, we wish to analyze the algorithm, not the language. In particular, the language should not introduce a bias in favor of one of the two MIMD models.

In order to accomplish this analysis, an algorithm must be stripped of its language dependencies. This can be accomplished with a generic set of MIMD operations. Every language construct must map to an equivalent MIMD operation or set of MIMD operations. These operations will identify a global memory access/communications operation. From this common intermediate form, a cost model for performing the required access on a particular architecture can then be applied. By mapping the high level language constructs to this intermediate representation, we can migrate the analysis away from language dependencies and towards the relationship between the algorithm and the target architecture.

Concurrency Control

Two aspects of concurrency control are the specification of when processes can proceed concurrently and the converse operation of preventing (presumably harmful) simultaneous access to shared resources. In relating these concepts to the performance of an algorithm on an MIMD architecture, it is profitable to focus on the fundamental mechanisms by which concurrency is regulated. In the next section, the use of the semaphore as a viable concurrency primitive for use in algorithm analysis is outlined from two points of view: (1) Semaphores can be used to express the synchronization/concurrency indicated by higher level language constructs. Thus, independent of language, algorithms can be mapped to a representation which uses semaphores. (2) Semaphores can be implemented on both Shared Memory and Private Memory machines. Thus the algorithm can be evaluated with respect to different architectures.

Semaphores

Concurrency specification is another facility with many forms of expression. A partial list of mechanisms in use includes semaphores[17], test and set[18], guarded commands[19], replace-add[20], fetch and add[1], fetch and ϕ[21], path expressions[22], interface modules[13,14,15], fork/join[18], cobegin[23,24,25], monitors[26], rendezvous[9], event counts and sequencers[27], channels[16], and messages[28].

Obviously, there are many ways to express synchronization in algorithms. To allow analysis, the goal is to identify a common form which can describe any of these synchronization mechanisms. This form should be close to hardware implementations so that further algorithm analysis can correlate the algorithm with the architecture.

One of the oldest synchronization methods are Dijkstra's P and V operators[17]. These provide a basis for more modern proposals for synchronization mechanisms. Briefly, a semaphore is an integer valued variable which can have P and V operations applied to it. The V(S) operation increments the semaphore S in an *indivisible* fashion. The P(S) operation decrements the semaphore S when the result would be non-negative. The last test and subsequent decrement is an *indivisible* operation.

The classical use for semaphores is in regulating access to shared resources. However, the semaphore can also act as a low-level, "common denominator" notation for specifying concurrency. For instance, Edison[23,24,25] provides a **cobegin** statement in which a parent process creates any number of processes, then waits for their completion. The **cobegin** statement can easily be expressed using Dijkstra's notation. Suppose three subprocesses are all concurrently executing, or ready to execute given any scheduling constraints. Also suppose that the main process is running. This would require two semaphores, *start* and *end* , both initialized to 0. Fig. 1 shows the code for the main process and one of the subprocesses.

Path Pascal[22] provides another good example of a language that can be translated meaningfully to P and V notation. It has a very complicated syntax to describe the concurrency of the program. This involves path expressions which specify when processes can be invoked in relation to other processes. For instance the path expression

path proc1 ; proc2 **end**;

signifies that proc2 should run only after proc1 has completed. Any number of these sequences may be active at one time. Likewise the path expression

path 3:(beginproc; endproc) **end**;

```
process main
        Do initial processing;
        V(start);
        V(start);
        V(start);
        P(end);
        P(end);
        P(end);
        Do more processing;
end process

process proc1;
        P(start);
        Do useful work;
        V(end);
end process
```

Fig. 1. Edison Coroutine Example

signifies that endproc may only follow beginproc, and there may be up to three concurrent executions of this sequence. Concurrency limitations even as complex as Path Pascal's can be described with semaphores. The condition that proc2 must follow proc1 is insured by a semaphore S with initial value of 0, such that proc1 executes a V(S) at its end, and proc2 executes a P(S) before it begins. Likewise, to limit the number of currently active paths to N, a semaphore S is initialized to N. A P(S) is placed at the beginning of the path, and a V(S) is placed at the end.

It is important to note that the definition of a semaphore describes a behavior, not an implementation. As such, it is an appropriate construct for describing the concurrency-related characteristics of an algorithm. As discussed below, it is model-independent in the sense that there are implementations suitable for both Shared Memory and Private Memory machines, so an algorithm whose synchronization and concurrency requirements are expressed in terms of semaphores can be mapped to either model.

In the subsequent step of evaluating an algorithm with respect to a particular architecture, the implementation will be of interest. Commonly, the semaphore notation is associated with a variable in a global memory system. This follows the definition closely for the V operation. The implementation of the P operation is highly dependent upon the machine architecture and even the operating system. The definition does not specify what a process is to do while it is waiting on the semaphore. Depending on the circumstances, the process may loop continually testing the semaphore. Alternatively, the process may "sleep" and allow another process to use the same CPU. In this case, the P operation is responsible for "waking up" the "sleeping" processes. The sleeping and waking is often accomplished by intervention by the operating system. The second implementation is prevalent in single CPU time-sharing systems. These implementations seem suited for a Shared Memory machine. The major problem is guaranteeing the mutual exclusion during the indivisible operations.

Likewise, there are implementations of P and V best suited for a Private Memory machine. In one implementation, a single PE would store the semaphore in its private memory, and keep a queue of P requests, then respond to the Ps whenever a V is performed on the same semaphore. The example in Fig. 2 uses this method. A single PE has exclusive access to each semaphore and any other PE must communicate with that one PE to gain access to the semaphore. This guarantees the mutual exclusion needed for the semaphores. This implementation would well take advantage of a separate control unit (CU) or PE to handle these actions. In Fig. 2, note that only the procedure "handle_events" can actually modify the semaphore. A simplified implementation for P and V is also shown in Fig. 2.

Here the CU has to reply to every P operation. The requesting PE waits until it receives the reply. No special restrictions need be placed on the accessing of the semaphore, since all the accesses are done by the single CU. If the value of the semaphore is greater than 0, a P operation is immediately acknowledged with a reply. When the PE receives the reply, it continues its execution. When the semaphore is less than or equal to zero, the CU keeps track of Ps by keeping a queue for each semaphore containing the PEs that have performed

```
procedure P(semaphore)
    send_message_to_CU(PE, P_MESSAGE, semaphore);
    wait_for_reply_from_CU(PE, semaphore);
end

procedure V(semaphore)
    send_message_to_CU(PE, V_MESSAGE, semaphore);
end

procedure handle_events
    read_message_from_net(PE, message, S);
    case message in
      P_MESSAGE:
       if S.semaphore > 0 then
            send_reply_to_PE(PE, S);
            S.semaphore = S.semaphore - 1;
       else
            enqueue(event.queue, PE);
       end if
      V_MESSAGE:
       S.semaphore = S.semaphore + 1;
       if S.semaphore > 0 and
       NOT_EMPTY(S.queue) then
            dequeue(S.queue, PE);
            send_reply_to_PE(PE, S);
            S.semaphore = S.semaphore - 1;
       end if
    end case
end procedure
```

Fig. 2. P and V Implementations in a Private Memory System

a P on that semaphore and are waiting for a V from another PE. It puts the PE into a queue associated with the semaphore S through the routine enqueue(S.queue, PE). Likewise, it removes a PE from the queue associated with S through the routine dequeue(S.queue, PE). The queue length must be as large as the number of processes.

For both Shared Memory and Private Memory implementations, semaphore access can become a bottleneck. For specific algorithm/implementation environments, simulation can be used to assess this; for the more general case, statistical and queuing analyses can be applied. Techniques to avoid the bottleneck involve distributing the load. An example of this in a Shared Memory machine is the use of Fetch and Add hardware in the NYU Ultracomputer[1]. In a Private Memory machine, various PEs (rather than a single CU) can be responsible for semaphores. To allow a process to know which PE controls a given semaphore, the compiler could associate a simple tag with each semaphore. The distribution of the semaphores across the PEs would reduce the likelihood of bottlenecks, and the tagging would not add a significant cost to the implementation.

Extensions to Semaphores

In the previous section, the feasibility of using semaphores as a primitive for algorithm analysis was discussed, in terms of the mappings from both high level language to semaphore representation and from semaphore representation to architecture. With some minor extensions to P and V, more general mechanisms can be provided that more closely correspond to modern day architectures. In the Edison coroutine example, the parent process executes three V operations and three P operations. These operations could be done just as well with slightly expanded P and V operations called Pn and Vn. The Vn(S, N) operation adds N to the semaphore S in an *indivisible* fashion. The Pn(S, N) operation subtracts N from the semaphore S when the result would

be non-negative. The last test and subsequent subtraction is an *indivisible* operation. The cobegin statement in the Edison main process would simply be implemented as Vn(start, N) followed by Pn(end, N), where N is the number of parallel branches of the algorithm.

Pn and Vn can be used in this context as a superset of P and V since P(S)≡Pn(S,1) and V(S)≡Vn(S,N). Pn and Vn retained the meaning taken from the higher level constructs. In the Edision example, the Pn and Vn implementation is more direct than the P and V implementation. There is a trend to put higher level synchronization facilities in parallel languages and architectures. A set of notations which includes P and V, but which also includes more complex synchronization mechanisms may therefore be useful in analysis. A higher level notation must, however, be able to map directly and equivalently to the low level notation (P and V). Only then would it be applicable to an architecture which does not support the higher level capability.

In order to show that this extension to a larger set of primitives is valid, possible implementations of Pn and Vn are given. Pn and Vn can be implemented using only P and V as shown in Fig. 3. The semaphore S becomes a variable containing two parts. S.semaphore is a semaphore valued variable which contains the value associate with S. S.simple_semaphore is a semaphore which can have only P and V operations performed on it. S.simple_semaphore is initialized to 1. Then accessing S.semaphore is bracketed by P and V operations on S.simple_semaphore. This insures that only one process may access with S.semaphore at any time.

One must take care in blindly converting groups of P operations into a single Pn operation. A problem occurs if two or more processes have outstanding Pn operations on a semaphore with different values of N. The process with the largest value of N may be locked out by the other processes, since they can continue when the value of the semaphore reaches some value smaller than the largest value of N. Pn could be defined differently to account for this condition. In translating the joining of concurrent paths into a Pn operation, only one process may execute the Pn, so for this analysis this is not a problem.

Since many languages and architectures provide mechanisms for higher level synchronization constructs, it is desirable to use a higher level mechanism where possible. This higher level notation must satisfy the two following conditions: (1) P and V can be simply defined in terms of the mechanism. (2) The mechanism can be simply implemented in terms of P and V.

First, in any analysis, it will be necessary to recognize the P and V operation in its basic form so it can be analyzed in a consistent manner. Secondly, for those machines not supporting the provided constructs directly, the defined operations should be easily implemented using only common machine instructions and P and V operations.

```
procedure Pn(S, N)
    loop forever
        if S.semaphore − N ≥ 0 then begin
            P(S.simple_semaphore);
            temp = S.semaphore − N;
            if temp ≥ 0 then
                S.semaphore = temp;
            V(S.simple_semaphore);
            if temp ≥ 0 then
                RETURN;
        end if
    end loop
end procedure

procedure Vn(S, N)
    P(S.simple_semaphore);
    S.semaphore = S.semaphore + N;
    V(S.simple_semaphore);
end procedure
```

Fig. 3. Pn and Vn as Defined in Terms of P and V

A Generalized Semaphore Notation

It has been shown how the P and V operations can be extended to the more general Pn and Vn operations, based on the stated restrictions. There are numerous ways to extend the P and V operations to various forms. In developing the NYU Ultracomputer[1,21], a generalized notation was developed to describe Fetch and Add and other similar synchronization constructs. This notation will be borrowed, then extended. The extension shows how many general semaphore mechanisms can be defined in terms of two functions.

The Fetch and Add operation is defined as shown in Fig. 4. The part of the operation between the braces is considered indivisible.

```
FetchAndAdd(G, L)
    { Temp ← G
      G ← G + L }
    RETURN Temp;
```

Fig. 4. FetchAndAdd Definition

This is equivalent to Vn operation described earlier. Pn can be defined in terms of Fetch and Add as shown in Fig. 5[21].

```
procedure Pn(S, N)
    loop forever
        if S − N ≥ 0 then begin
            temp ← FetchAndAdd(S, − N);
            if temp ≥ N then
                RETURN;
            else
                FetchAndAdd(S, N);
        end if
    end loop
end procedure
```

Fig. 5. Pn in Terms of FetchAndAdd

There are many possible hardware facilities available to support similar operations. Rather than picking one facility as a basis for all synchronization mechanisms, a class of mechanisms is defined based on the two restrictions previously given. The goal is to define operations which correspond to P and V, but which can account directly for a wider variety of high level language constructs and hardware implementations.

The first operation corresponds to V, and is the FetchAndϕ operation proposed in Gottlieb and Kruskal[21]. It is defined as shown in Fig. 6.

```
FetchAndφ(G, L)
    { Temp ← G
      G ← φ(G, L) }
    RETURN Temp;
```

Fig. 6. FetchAndϕ Definition

Table 1. Common Uses of FetchAndϕ

Operation	Expression in FetchAndϕ
R ← TestAndSet(G)[2]	R ← FetchAndOr(G, TRUE)
R ← FetchAndAdd(L,G)[2]	R ← FetchAndAdd(G, L)
V(S)	* ← FetchAndAdd(S, 1)
Vn(S, N)	* ← FetchAndAdd(S, N)

ϕ is a suitably defined function, G is a variable, and L is a value. Only a few ϕs have proven themselves useful in synchronization statements. A few common ϕs are given in Table 1. A * in the expression indicates that the value is not used and need not be returned.

Logically, this notation can be extended to describe a class of operations that corresponds to the P operation. Thus, the notation is expanded here to include WaitForθ. This has two equivalent definitions/implementations based on the capabilities of the hardware. These are shown in Fig. 7. As before, G is a variable, L a constant. The new argument C is a condition that must be satisfied before the function will return. The first implementation assumes the hardware is capable of performing a test and conditionally performing the equivalent of FetchAndθ. The second insists only that the hardware be capable of the FetchAndθ operation. Note that the second form may perform unnecessary steps. It decides that the operation should probably succeed and then in another *distinct* step, it attempts the θ operation. In case its assumption was invalidated, it checks afterwards and "undoes" the θ function with θ^{-1}.

```
WaitFor θ(G, L, C(G))
    loop
        { if C(G) then
            Temp ← G;
            G ← θ(G, L); }
          if C(Temp) then
              RETURN Temp;
    end loop

WaitFor θ(G, L, C(G), θ⁻¹(G, L))
    loop
        if C(G) then begin
            Temp ← FetchAndθ(G, L)
            if C(Temp) then
                RETURN Temp.
            else
                FetchAndθ⁻¹(G, L)
    end loop
```

Fig. 7. WaitForθ Definitions

Table 2 shows some examples of synchronization operations that can be expressed in terms of WaitForθ functions. A binary semaphore can have values 0 or 1 (free or reserved). The Wait on a Binary Semaphore operation waits until the semaphore is free (0) and reserves it (sets it to 1) in one *indivisible* operation. The other constructs in the table are familiar. In all cases, the choice of θ, C and θ^{-1} functions must be consistent and must guarantee reliable operation.

A generalized notation for basic synchronization mechanisms has been presented. The corresponding examples in Tables 1 and 2 show four pairs of mechanisms that can be described in terms of the generalized notation, including P and V themselves. This model unifies several of these types of mechanisms into a common notation. These mechanisms may be realizable in hardware for some architectures. They can safely be used in analysis, since they can also be realized using only P and V, the basic semaphore operations. It is desirable to have a number of these mechanisms available for analysis since software specifications of these mechanisms can be mapped to the most natural hardware implementation, rather than to a less obvious and more artificial implementation.

A Simple Set of Language-Independent MIMD Operations

An MIMD algorithm will consist of portions that execute simply as serial code on a single PE along with several operations specific to MIMD operation. The principal such operations have just been described. Throughout the discussion, it has been argued that the common language operations map into a few simple generic operations. The operations map as given in Table 3. Attempts to map features into more complex operations result in counter-examples of constructs from some languages that will not map into the low level mechanism.

Table 3. Mapping to Generic Operations

High level operation	Low level operation
Arithmetic/Logical	Same as serial analysis
Conditional Branching	Same as serial analysis
Global Variable Access	Communications/Memory
Interprocess Communications	Communications/Memory
Concurrency Control and Synchronization	P/V or FetchAndϕ/WaitForθ

Traditional analysis techniques exist for enumerating the time/space costs of simple arithmetic statements and loop control statements. The most significant problems with MIMD algorithm analysis stem from global variables/communications (GV/C), concurrency, and synchronization. Because of the generality of the GV/C and concurrency/synchronization primitives shown here, it is possible to map the high level constructs from a wide variety of languages into combinations of these primitives. Thus, in analysis, the occurences of any given high level construct would map to a series of these generic operations.

Example

A simple example of an analysis using the proposed primitives is presented. Although the example is very simple, it illustrates the mapping of the high level language algorithm to a set of primitives. The example is

Table 2. Common Implementations Using WaitForθ

Operation	result	ϕ	G	L	C(x)	ϕ^{-1}(G,L)
Wait on Bin. Sem.	*	OR	G	1	x = 0	G
P(S)	*	Subtract	S	1	x ≥ 1	G+1
Pn(S, N)	*	Subtract	S	N	x ≥ N	G+N
R ← WaitAndSub(G, L)	R	Subtract	G	L	x ≥ L	G+L

Paper 12

Models for Use in the Design of Macro-Pipelined Parallel Processors

From the Proceedings of The 12th Annual International Symposium on Computer Architecture, June 1985, pp. 116-123.

MODELS FOR USE IN THE DESIGN OF MACRO-PIPELINED PARALLEL PROCESSORS

Bradley Warren Smith and Howard Jay Siegel

PASM Parallel Processing Laboratory
School of Electrical Engineering
Purdue University
West Lafayette, IN 47907

Abstract

An approach is proposed for modeling off the shelf hardware and for modeling parallel algorithms, along with a design methodology to use the information provided by these models, to design a class of macro-pipelined special purpose architectures. Nine parameters to form a model of the characteristics of parallel/distributed algorithms and the environment in which they must execute are presented. In addition, a set of tuples to model the characteristics of computer architectures is presented. By combining the tuples with the parameters, the execution time of the algorithm modeled by the parameters on the hardware modeled by the tuples can be approximated. The combination of these models could be used as a basis for computer aided tools used in the design of macro-pipelined parallel/distributed processors.

1. Introduction

For certain applications, such as speech processing, time is an important factor. In such applications, there is a need to process many data sets in the same way e.g., continually performing an FFT for every frame of input data. Previous analysis, such as that performed in [4], [5], [34], [35], and [37] indicate that for many types of tasks, conventional general purpose processors are insufficient. In this paper, an approach is proposed for modeling off the shelf hardware and for modeling parallel algorithms, along with a design methodology to use the information provided by these models, to design a class of macro-pipelined special purpose parallel architectures. The ultimate goal is to use models such as the ones proposed here to develop computer aided design tools. Special purpose processing systems (such as those used for dedicated real-time analysis) are typically sold in small quantities. As a result, the cost of the design can make the resulting system prohibitively expensive. Computer aided design tools for this process would reduce the cost involved and are therefore desirable.

This paper uses nine parameters to correlate the hardware to be designed with the applications software to be executed and the I/O environment in which the machine will operate. A macro-pipelined layered approach to task decomposition is demonstrated. Each portion of the decomposed task is then assigned to a special purpose processing unit. This implies that each processing unit may either be a traditional serial type design or a parallel design. Once this initial decomposition is established, techniques such as those used to adjust the execution time and throughput of a pipeline in [14] can be applied.

In this approach to reaching the goal of computer aided computer design, functional descriptions (models) of the hardware components that may be used in the design must be combined into a database. Included in such descriptions are information about the cost of the device, an enumeration of all the operations that it can perform, and the pathwidth and execution times for those operations. More complex taxonomies, such as those found in [7], [9], [10], and [12] are not needed for the database because they specify architectural information that is unneeded here.

The information in the database will be used to select hardware to perform a given task within cost and time constraints. Time constraints are in two forms, response time and throughput. *Response time* is the time between receiving an input and completion of the corresponding result. *Throughput* is the number of data sets processed per unit time.

Consider a task that is composed of several sub-tasks. An example of such a task might be isolated word recognition [17], [19], [23], and [37]. For isolated word recognition, a typical processing scenario might be: digital filtering, autocorrelation analysis, linear predictive coding (LPC) analysis, linear time warping, and dynamic time warping. Each of these processes (sub-tasks) represents a portion of the scenario. Each of these sub-tasks will be called a *layer*. Using information about each sub-task a special-purpose architecture can be developed to perform the sub-task within some time and cost constraints. The special-purpose hardware that is assigned to each layer will be called a *level*.

For simplicity, only scenarios in which there is no feedback will be considered. Initially, the layers will be chosen according to conceptual differences, i.e., digital filtering is different from autocorrelation analysis, so each should be a different layer. This uses the simplifying assumption that conceptually different portions of the task (the layers) will require different hardware resources to produce an initial configuration. The layers and their associated levels of an isolated word recognition system are shown in Fig. 1.1.

It is the goal of this scheme to achieve a higher throughput by decomposing a scenario into layers. Because each layer requires fewer computations than the entire scenario, connecting the levels in a macro-pipeline and pipelining the data sets through the machine should increase the throughput of the resulting system. This type of parallelism is referred to here as *vertical* parallelism. Furthermore, each layer is executing on specially designed hardware, which may employ multiple computational units, so the response time of the resulting system is decreased. The parallelism occurring within a given level, where multiple units are performing operations on different portions of the data set simultaneously, is referred to as *horizontal* parallelism. This vertical and horizontal parallelism is similar to the techniques of subdivision and replication discussed for pipelines in [14] or the "purely pipelined" and the "purely parallel" architectures discussed in [36]. Throughput constraints may require that a layer must be further divided into smaller processes. These will not represent new layers, but *sub-layers*, which will correspond to *sub-levels* of hardware, consistent with the previous nomenclature.

By developing a model to transform a task description i o a potential macro-pipelined architecture, a machine can be built with the necessary characteristics to execute the task quickly and without excessive amounts of hardware. A basis for such a model is proposed and analyzed in this paper. The information provided by the nine parameters mentioned earlier will allow each level to be designed for a specific sub-task, having a special hardware complement to perform that sub-task more quickly. Each level can use SIMD and/or MIMD [6] parallelism. The

This research was supported by the U.S. Army Research Office, Department of the Army, under Contract No. DAAG29-82-K-0101, and by the National Science Foundation, under Grant No. ECS-81-20898.

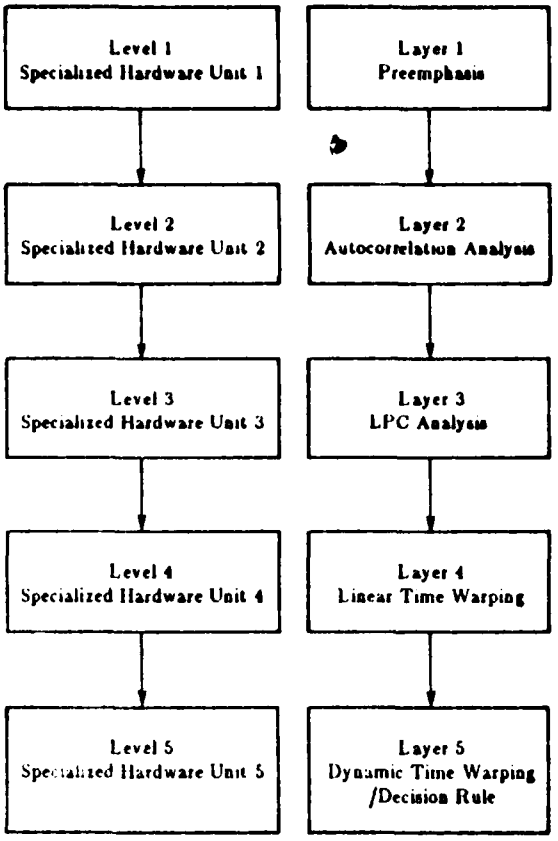


Fig. 1.1 Layering of Isolated Word Recognition System

result of the technique is to design a machine that can perform a processing scenario within some time constraints.

It is the goal of this paper to introduce methods of modeling hardware and algorithms so that an reasonable approximation of the execution time of an algorithm on a special-purpose system is possible. The hardware model is discussed in Section 2. An overview of the proposed design scenario is presented in Section 3. In addition, types and limitations of various forms of parallelism are discussed. Section 4 presents nine parameters that model an algorithm and discusses the calculation and significance of each of the parameters. An example of the design methodology is given in Section 5.

2. The Hardware Database

A processor description in the database consists of two 6-tuples, two N-tuples, and three N+1-tuples, where N is the number of assembly language instructions (the "+1" is used to describe the instruction fetch unit). The first 6-tuple consists of the processor name, cost, clock speed, data pathwidth, address pathwidth, and virtual address width. The second 6-tuple consists of the size and speed of on-board cache, the size and speed of on-board memory, and the size and width of the registers. The N- and N+1- tuples must be able to answer questions regarding the execution time for all processors in the database. Thus, the tuples must provide information about the type of machine instructions, the execution time for a single operation for each instruction, the number of stages in any pipelines, the replication of units, and the overlap of operations. The tuples corresponding to the last three information categories are N+1-tuples to account for any pipelining, functional overlap, and parallelism that can occur within the instruction fetch unit. By combining the information contained in the various tuples, it is possible to determine the exact calculation time of all operations whose times are constant. For example, by combining the number stages in a pipelined unit with the single operation execution time of the unit, it is possible to determine the throughput of the unit.

Because different processors have different instruction sets, N is not the same for all processors. Consider the case of a simple processor with an instruction set consisting of an 8-bit add, a 16-bit add, a return on zero, a move memory to register (8-bit), and a move register to memory (8-bit). The first 6-tuple would look like:

(BRAND/MODEL, $5.00, 1.3$\mu$sec, 8-bits, 16-bits, 16-bits)

The tuple describing the type of machine instructions would look like:

8-bit add register to register
16-bit add register to register
return if zero
8-bit move memory to register
8-bit move register to memory

For this tuple, both the source and destination must be enumerated. This allows for processors like the 8085 in which registers can only be added to the accumulator.

The other tuples contain the types of information mentioned earlier, where information in the i^{th} element corresponds to the i^{th} instruction. By including this information in the database, it is possible to recreate the timing information stored in the architecture description set forth in [12].

For the purposes of this paper, the units considered for the database are either single chips or small boards. The underlying assumption for this scheme is that there is no shared or reconfigurable pipeline units on board. When this assumption becomes false, two N+1-tuples will be required to represent shared pipelines and their reconfiguration times. Other factors that should be included in the data base are power consumption, heat dissipation, and size. While these last three factors do not influence performance, they do provide necessary application information about the possible environment in which the chips can operate.

A functional description, such as that found in [2], can be used to accurately categorize each unit according to its functional capabilities. To this point, only processing hardware has been considered. The hardware database can be divided into the functional units of processor, memory, input/output, vector, and array processors. This is consistent with [2]. Each functional unit will have its own set of tuples used to describe its performance. The tuples will be used with the characteristics of the application algorithm to choose specific hardware for each level of the system.

Included with the hardware descriptions of the processors in the database would be a routine that can simulate that processor. By combining the simulation procedures with the architectural information of other components in the database, e.g., memories, it is possible to create a simulator for the proposed macro-pipelined architecture. Such a database with simulation routines for each relevant component would be a useful tool for the research community interested in the design of macro-pipelined special purpose systems. These tools would be used according to the approach presented in the next section.

3. The Design Scenario

After the initial layering is performed, an exact statement of the application algorithm to be performed at each level is needed. This is done using nine parameters that are discussed in the next section. This information is then used in conjunction with the hardware description to evaluate the performance of each processor in the hardware database. Then information about the desired throughput and average desired response time ($\bar{T}_{r_{des}}$) of the system must be gathered. These will be the evaluation criteria, i.e., can a proposed system process the data with the desired throughput and response time.

The first step in the modeling process is to choose all levels to process their incoming data as fast as possible without using vertical or horizontal parallelism within any given level. Since

this type of design is a macro-pipeline, the throughput of the pipeline is limited by the slowest level.

Macro-pipelined architectures produce a continuous flow of data. The time to process a single data set (the time for data to go from the first level to the last level, i.e., the response time) in such a vertical architecture is same as for a single-processor serial system because the data must be processed by the multiple levels of hardware. The throughput for multiple data sets is greatly increased because new results are completed at a rate equal to the processing time of the slowest level or sub-level. If the time to go from the first level to the last level is too slow, since the levels are designed with the fastest serial processors in the database and only off the shelf parts are allowed, horizontal parallelism, such as that found in SIMD or MIMD machines, must be applied. For example, if the processing time for all levels and sub-levels of an architecture were halved, the time to go from the first level to the last level would also be halved. Thus, vertical parallelism can be applied to increase throughput, while horizontal parallelism can be applied to increase throughput and decrease response time.

If the required throughput is 1 job/T_T seconds, than each level must execute its layer in at most T_T seconds. If the machine fails to meet the throughput qualification, the execution speed of all levels not meeting the time constraint (T_T) must be increased. This can be accomplished with the previously discussed horizontal and vertical parallelism.

The maximum amount of horizontal parallelism that can be applied to a task is the inherent parallelism of the subtask to be performed (the minimum horizontal parallelism at any level is a single unit). Further, horizontal parallelism is affected by precedence constraints of the subtask. Typically, each additional processor used for horizontal parallelism will not increase the execution speed linearly, i.e., the speedup may be less than a factor of P using P processors for any P. This is discussed in [31]. The minimum vertical parallelism is one processor and the maximum vertical parallelism is up to one processor per instruction. Using one processer per instruction will not only cause a potential architecture to be prohibitively expensive, it may require an exorbitant amount of overhead to implement. Vertical parallelism is not affected by precedence constraints because they are still enforced; however, vertical parallelism will not reduce the response time. Thus, there are associated costs and limitations with both vertical and horizontal parallelism.

There are two additional limitations on the type and amount of parallelism applied at each level. The first is that there is an upper bound on the cost. An additional limitation is placed on the type and amount of parallelism by requiring that all parts be "off the shelf." This second limitation forces the architecture to be buildable with present day technology. These limitations assume that an algorithm can be structured for horizontal parallel execution. If an algorithm is unsuitable for horizontal parallel execution, vertical parallelism will be required.

It is required that there be some form of coordination between the levels. This can be either (a) a master system clock that tells each level when it can proceed to the next data set or (b) a unit that keeps track of all levels and, when all levels are done, signals each to proceed to the next data set. A system executing with a master clock will typically execute more slowly than a system where each level reports its status to a control unit. If T_i is the time required for level i to complete its subtask given its current data set, then the master clock cycle time T_C must be set to the maximum value of T_i over all levels for all data sets. The implementation suggested in (b) for an L level system will require an execution time T_e of: $T_e = \max(T_1, T_2, T_3, \cdots, T_L)$. There is additional overhead for scheme (b) in terms of control hardware and signaling time. Thus, if it is expected that there will not be a significant difference between T_C and T_e, method (a) would be preferable. In the extreme case, $T_C = T_e$. Normally, T_e will be much less than T_C.

To fully utilize the hardware in the system, it is desirable to match the speed of all the levels. This can be done in an L level system by forcing the average response time of level i, $\bar{T}_i$, to be $\bar{T}_i = \bar{T}_{rdes}/L$. After the initial design (all levels designed to perform their layer as fast as possible with no vertical or horizontal parallelism), the data processing rate of all the levels will be known. If the designed machine meets or exceeds the throughput and response time qualifications of the scenario, faster levels that are adjacent can be combined. Faster levels can also be built with slower and less expensive hardware. This will still maintain the throughput of the system, however, the response time of the system may be increased. Such a process can be repeated as long as the throughput/response time requirements are met. This will lower the cost of the overall system.

To propose and evaluate candidate architectures for levels, a mapping is required between a layer and its corresponding level. Included in this mapping is the description of the layer in terms that relate it to the computational requirements that it places on the hardware. It is this mapping that is the topic of discussion in the next section. Using information from the hardware database discussed in section 2, the performance of candidate architectures can be evaluated by some measure such as those in [28].

After the architecture of all the levels have been proposed, the approximate performance of the system is known. Simulation is required for an exact evaluation of the performance of the system. This is required to insure that the system will perform as desired.

4. Nine Evaluation Categories: Their Relationship to Hardware and Software

When designing hardware for a specific algorithm, characteristics of the algorithm must be "mapped" onto the hardware. To build hardware to execute a given layer, a user must supply each of the of the following *evaluation parameters* about each layer in the system.

(1) Type, rate, and amount of input
(2) Type and number of operations per input datum
(3) Range and accuracy of arithmetic data to be used
(4) Algorithm to be used
(5) Type, frequency, and message length of processor-to-processor communications
(6) Amount of memory required
(7) Type, amount, and benefit of parallelism
(8) Type, rate, and amount of output
(9) Evaluation criteria

These parameters form a model of the algorithms in the task. The information they supply can be used with the hardware model of Section 2 and the design scenario of Section 3 to develop a macro-pipelined architecture for the task.

Category (1) places restrictions on the input buffering, input data rate, and the internal data format of a level. The type of data specifies the format and word width required to process the incoming data. Combined with the rate, the type of data specifies the speed of the input unit. Between levels, either double-buffering or triple-buffering [5] may be used; i.e., two or three memory units may be employed to between adjacent levels to allow the overlap of computation and I/O. If the application does not require real-time processing, then the system must be such that the incoming data rate of the first level determines the steady state throughput. For real-time applications, the incoming data rate of the first level determines the minimum data rate for the system. The difference is: a non-real-time system can stop the incoming data stream as necessary; however, a real-time system may not be able to stop the incoming stream of data without losing data.

Evaluation category (2) determines the specific number of operations that must be performed by a given level in time T_i. From one data set to another the required processing may vary, so an exact statement of what operations must be performed may be unavailable; however, a reasonable estimate may be calculated for either the average case or the worse case through either simulation techniques or statistical analysis, as was done

in [29]. Depending on the application of the hardware, either an average value for the number of calculations or a worst case value may be used.

The number of each operation can be multiplied by the corresponding execution time for a single instruction (from the first N-tuple). If the resulting values are summed and multiplied by the clock speed of a processor, the worst case execution time can be determined for that processor. To yield a more precise bound on the execution time of a process with processors that contain pipelines or parallel units, either simulation or task analysis such as that shown in [16] must be applied to the algorithm.

Classes of algorithms of concern for this parameter are those algorithms that perform the same operations on each data element (data independent) and those algorithms that treat each data element differently (data dependent). For data independent algorithms, the number and type of each operation performed are countable from the algorithm. For a data dependent algorithm, the number of operations can be determined through simulation on sample data sets or, in some cases, through analysis making certain assumptions about the characteristics of the data. Typically, data dependent algorithms require varying resources and processing times.

The Data Dependency (DD) of an algorithm is:

$$DD = \frac{\text{Data Dependent Operations}}{\text{Total Operations}}$$

and can be used as an indicator of what percentage of the expected execution time is fixed (i.e., data independent) and what percentage may vary (i.e., data dependent). It also indicates the appropriateness of SIMD or MIMD parallelism.

Operations can be divided into five groups: (A) arithmetic and logic, (B) addressing, (C) index calculation (loop variables), (D) conditional, and (E) inter-processor data transfer. These classes were chosen to yield partial information about which operations can be overlapped. For example, on some SIMD systems, operations in (C) and some in (B) can be done in the control unit, overlapped with the parallel execution of the rest of the operations done by the processing elements. Information about class (E) indicates how much the network will be used. On a system where all processing is done by the same unit, the distinction between the types of operations is diminished; however, to construct special purpose hardware for real-time processing, the distinction is useful.

Information about the (A), (B), and (C) must be further sub-divided to provide information necessary to choose suitable processing hardware. For example, (A) and (C) should be divided into: floating point additions, subtractions, multiplications, divisions, comparisons, and special functions; and fixed point additions, subtractions, multiplications, divisions, comparisons, and special functions. (B) should be divided into load and store.

The number of operations in each of the above sub-groups gives the absolute number of each operation to be done. From this, it is possible to calculate the relative importance of the speed of each operation. For each floating point or fixed point special function, the number of times each operation is expected to be performed is specified along with an *equivalence relation*, giving the number of "standard" [11] operations needed to implement the specified function in software. If a unit cannot perform a specified function directly in hardware, the time required to synthesize that function (specified by the equivalence relation) must be calculated. If a special device (e.g., coprocessor) is available to perform the special functions, the need for including this device can be determined. By using this approach, various units can be ranked by their execution speed for a given algorithm.

The numerical range and accuracy (3) places various limitations on the hardware. Typically, more accurate hardware (larger words) will be slower and/or more costly than hardware with smaller words. Thus, it would be advantageous to use the smallest word size meeting the range and accuracy constraints. Floating point operations are typically slower than the corresponding integer operations. In certain cases, if the numerical range required for various calculations is small, but out of the range of specific hardware, e.g., underflow, normalization of data can eliminate the need for special hardware at the cost of some processing time. The arithmetic range associated with a set of operations greatly affects the hardware required [30]. Approaches to dynamic word size machines, such as those in [1], [15], and [18], can be employed in cases where arithmetic ranges vary from loop to loop.

The numerical range and accuracy of a sub-task is a function of algorithm and data. For an algorithm, it is necessary to determine the maximum and minimum values of the range of the calculations. The range of the calculations should be divided according to the range of index values, range of integer arithmetic, and the range of floating point arithmetic. This specifies, in the SIMD case, the word size of the control unit, and the word size of the integer and floating point units. In other cases the word size of the integer unit is set according to the maximum range required for integer and indexing arithmetic.

With knowledge about the algorithm a level is to process (4), SIMD and/or MIMD horizontal parallelism can be introduced. Special parallel analysis techniques, such as those discussed in [3], [16], and [22] can be employed to utilize "extra" parallelism. This can be accomplished by breaking the algorithm down into multiple streams, using MIMD parallelism. Applicable loops are those containing variables that can be calculated independently of other variables within the loop. The "breakdown" occurs when a variable can be extracted from a loop and calculated *in a separate environment (either a different processor or processors)* [3]. Other techniques for parallel processing such as the use of "recursive doubling" for calculating sums or maximums [32] using SIMD or MIMD parallelism can be applied.

The algorithm is required to obtain timing information from the previously discussed N-tuples describing the hardware database. By multiplying the number of each type of operation by the corresponding operation time, an upper bound on the execution time can be obtained. The algorithm must be scanned to determine what percentage of the operations can be pipelined and/or overlapped. This must be done for each processor in the database. After the amount of time saved by the parallelism and pipelines is determined, this time is then subtracted from the execution time for the processor. For systems with reconfigurable pipelines, the reconfiguration time must be multiplied by the number of reconfigurations required by the algorithm.

By deriving bounds on execution time as described in [13], levels requiring large amounts of time can be analyzed. This will indicate where each level is spending its execution time. If consistent variable names are used from layer to layer, similar task decomposition to the above can be applied across levels to allow the combination and/or sub-division of levels as needed. Consider the scenario in Fig. 4.1. If level three calculates a, b, and c independently of the output of level two, and the throughput of level three is too low, the portion of the algorithm calculating a,b, and c can be moved to level 2. If this makes the throughput of level two too low, a separate unit can be employed for the calculations. The result is shown on the right of the figure.

The type, frequency, and message length of the processor-to-processor communications within a layer (5) will dictate the topology of a level and the design of the interconnection network. There are two types of interconnection networks. A global interconnection network allows a given processor to communicate directly with any other given processor within a given horizontally parallel structure (e.g., SIMD or MIMD portion of machine). Typically a multistage arrangement is used for such a network [26] (although it does not permit al possible SIMD data permutations). The second type of interconnection network is local interconnection network, which allows a processor to communicate with a specific number of its neighbors (e.g., 4- or 8-nearest neighbors) [30] and [33]. In this case, the processors can be viewed as either a one, two, or three dimensional array when determining the connections to be made by the network. A network must be capable of making the desired connections efficiently and with minimal collisions, to avoid significant

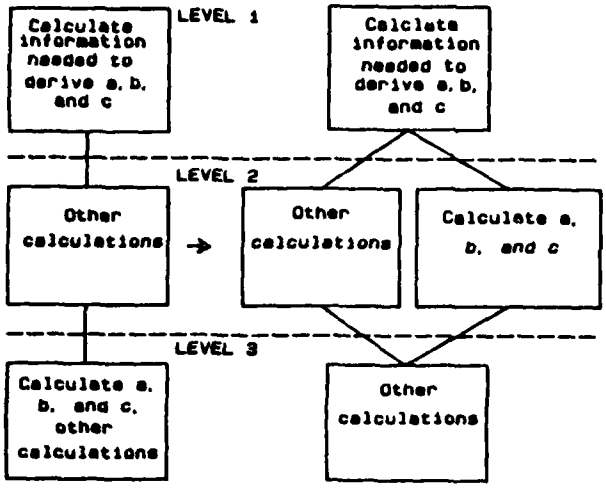


Fig. 4.1 Scenario before and after application of techniques in [13]

delays during transfers. It would be desirable to have a database of known global connection networks and the permutations that they can perform, so an appropriate connection network can be chosen.

From the type of communications required by a layer, information can be gained about the type of processing that should take place on a given level, i.e., the more random the communications, the more likely that a horizontally parallel level should use MIMD (asynchronous) parallelism, as opposed to SIMD (synchronous) parallelism. Knowing the length of the transfers will aid the design of the network. For instance, the longer the transfers, the more suitable a circuit switched network becomes. For small transfers, a packet switched network is desirable. The number of network transfers and the length of the average transfer provides information about the loading of a network with a given transfer speed.

Determination of the type and amount of processor-to-processor communication for a highly data independent task is straightforward and can be obtained from analysis of the parallel structure of the algorithm statement in (4). For data dependent tasks, the required transfers may vary in length and connection, dependent solely on the data set being processed. Simulation may be required to achieve accurate estimates. To minimize the need for simulation, analysis of the data set can yield information about the required connections. For example, if a process performs edge tracing on an image containing small objects (relative to the image size), global connections are not required, only local (nearest neighbor) connections are needed [34]. If the objects are large, then global connections may be needed.

Memory size (6) is an important factor in the design of a system and is a function of the proposed data set size, data type, and algorithm. The data set size, processors in a level, and algorithm chosen have an important bearing on how much memory is associated with a processor in a given level. This will be considered in addition to the buffer memory associated with a given level.

Memory usage falls into three classes: program, stack, and data memory. Program memory is not determinable from the algorithm, although an estimation is possible. It is a function of the machine and the compiler. The stack memory contains arguments to subroutines, return addresses, and temporary information. It is a function of the nesting of subroutines, along with the information that is passed to the subroutines. For data dependent recursive algorithms, simulation may be required to determine the appropriate amount of stack memory needed. An alternative to simulation is to place a maximum depth (in terms of calls to specific functions) on the stack. If each specific function is called with given arguments (each with a given size), calculation of the stack size is straightforward, based on the maximum number of calls times the space needed for each call.

The data memory size is composed of the index memory size and the process data memory size, where the index memory is the memory required to store loop counters and some index variables. The process data memory is the memory used to store the parameters, working data, intermediate results, and index variables that could not be stored in the CU of an SIMD system. For a data independent task, data set size is trivial to determine from a algorithm. A data dependent task may require simulation.

The particular divisions of memory stem from where the data must be accessed. In an SIMD environment, the stack, index memory, and program memory must be associated with the control unit, while the process data must be accessible by the processing elements. In other environments, this memory is associated with the processor, so the divisions do not matter so much as their total.

The type and amount of parallelism (7) will specify the nature and maximum number of processors associated with a given level. The benefit due to parallelism is specified in two areas: (I) speedup due to P processors and (II) the maximum value of P.

The type of parallelism is a function of the algorithm. Certain algorithms may be written for an SIMD machine, thus SIMD parallelism should be used. For a general algorithm, determining whether an algorithm is best suited for a specific environment can be done by looking at the DD, as discussed above. For a typical parallel algorithm, the lower the DD, the more likely an algorithm is suited to SIMD type processing. Typically, MIMD parallelism is more flexible; however, SIMD parallelism has the advantages of built-in synchronization and the ability to overlap CU control operations with processing element instruction execution.

The amount of parallelism can be determined by several criteria. Typically, the larger the number of processors, the less processing each processor must perform and the more significant transfer and wait times become. As transfer and wait times become more significant, the processors will spend a larger portion of time idled, so the utilization of a processor will decrease. A variety of performance measures are discussed in [28]. These can be used to determine the relative benefit of each additional processor, allowing one to calculate the number of processors associated with a given level.

The speedup due to P processors (I) can be obtained by analyzing the algorithm. This figure can be used to determine the decrease in response time by using P processors. The maximum value of P (II) is the ceiling on the amount of parallelism. This represents the maximum amount of inherent parallelism in a given task and can be calculated by analyzing the task. For both I and II, data dependent tasks simulation may be required.

Knowledge of the type, rate, and amount of output (8) will be required for any formatting that must be done to interface the data to the device gathering the results. In addition, it places constraints on the output data rate.

Finally, the evaluation criteria (9) define how the merit of a system is to be calculated. Here, the evaluation criteria will be speed and cost, i.e., the execution must occur in real-time for the minimum cost. For non real-time systems, other criteria such as those considered in [8] and [28] may be used, e.g., efficiency, utilization, and power consumption. By incorporating the evaluation criteria into the design procedure, proposed designs not meeting the evaluation criteria can be avoided. In addition, this provides a way to rank various designs.

5. Example of Approach

Consider the application of the nine parameters to a task such as Dynamic Time Warping (DTW), which is performed in speech processing. This algorithm warps incoming utterances to find the best match in a list of templates of known words. It represents the most computationally intensive portion of the proposed speech processing scenario and corresponds to one layer of the task (Fig. 1.1).

```
hold = ∞ ; template =0;/* initialization */
for k = 1 to 10000 {  /* for each template */
    for j = −1 to 1 { /* initialization */
        for i = −1 to 1 {
            g[i][j]=∞;
            d[i][j]=∞;
        } /* end i */
    } /* end j */
    for j = 1 to 80 { /* for each frame in a and b[k] */
        for i = j−r to j+r { /* each frame within window */
            if(i≤0) i = 1; /* force i to be valid */
            if(i>80) i = j−r+1;
            else {
                d[i][j]=0;
                for h = 1 to 9 {
                    /* compute 'distance' between
                      frames a[i] and b[k][j] */
                    d[i][j]= d[i][j] +
                        (a[i][h]−b[k][j][h])**2;
                } /* end h */
                g[i][j] = min(g[i−1][j−1] + 2d[i][j],
                    g[i−1][j−2] + 2d[i][j−1] + d[i][j],
                    g[i−2][j−1] + 2d[i−1][j] + d[i][j]);
            } /* end i */
        } /* end j */
    D(a,b[k]) = g[80][80];
    if(D(a,b[k]) < hold) { /* store minimum value */
        hold = D(a,b[k]);
        template = k;
    } /* end if */
} /* end k */
```

a	- unknown word (UW)
a[i]	- frame i of UW
a[i][h]	- element h of vector describing frame i of UW
b[k]	- reference word k (RWK)
b[k][i]	- frame i of RWK
b[k][i][h]	- element h of vector describing frame i of RWK
D(a,b[k])	- distance between UW and RWK
g(i,j)	- cumulative distance between a and b[k]
hold	- distance number of best fitting reference word
template	- number of best fitting reference word

Fig. 5.1. Sample DTW algorithm

A DTW algorithm is shown in Fig. 5.1 [37]. The input to this algorithm consists of 80 frames of speech, each represented by vector of nine 16-bit integers (80). There will be one 16-bit quantity used to identify each word. Assume there are 10,000 templates in the database (meaning the system can understand 1000 words since ten templates are required for each word [25]). The variable "r" is the amount the algorithm will be allowed to warp the incoming template. For the purposes of this paper, r=3. The nine evaluation categories are as follows:

I. Type, rate, and amount of input

Type: Fixed point data
Rate: 1 utterance/1.0 second
Amount: 720 fixed point numbers/utterance

II. Type and number of operations/input datum

6.8M index variable assignments
0.1M index variable additions
66.1M index variable additions (+1)
67.3M index variable conditional branches
132.7M address calculations
105.5M fixed point additions
5.8M fixed point assignments
11.3M fixed point conditional branches
60.7M fixed point multiplications
60.7M fixed point subtractions

III. Range and accuracy of arithmetic data

$\pm 2^{15}, \pm 1$

IV. Type, amount, and frequency of processor-to-processor communication

Type: Global, capable of recursive doubling [32]
Amount: 2 fixed point numbers
Freq.: 2 $\log_2 P$ transfers per second

V. Amount of memory required

Memory: (14.5/P)+0.01 Mbytes of data per processor for reference (template) and incoming utterance storage
10 Kbytes of program and stack
Note: one copy of the program is required per processor for MIMD machine; one copy in the control unit for SIMD machine.

VI. Type, amount, and benefit of parallelism

Type: SIMD or MIMD
Amount (max): 10,000 (utterance in database)
Benefit:

$$\text{speedup} = \frac{T}{\frac{T}{P} + \left\lceil (\log_2 P) \right\rceil (IC + 2 \times NO)}$$

where a single processor takes time T, IC is the time for an integer comparison, and NO is the time for a network operation.

VII. Algorithm to be used

Algorithm: See Fig. 5.1.

VIII. Type rate and amount of output

Type: 1 English word
Rate: 1 per second
Amount: 100 characters maximum (arbitrary)

IX. Evaluation criteria

Speed and cost

These nine evaluation categories represent an analysis of the algorithm. Evaluation category II is directly determinable from the algorithm. The range and accuracy is determinable from the application. [24] states that $2^{15} \pm 1$ is a reasonable range and accuracy for this task. To apply a parallel machine to this algorithm, each processing element would need to execute this algorithm on its own portion of the template database computing a local D(a,b[k]). Recursive doubling [37] would then be used to combine the results; i.e., the word associated with the smallest d(a,b[k]) is the chosen word. This requires $2\log_2 P$ transfers for the d(a,b[k])'s and the identifiers for their associated words.

The amount of memory is expressed as a function of P, the number of processors. A "C" language program was coded and compiled to estimate the program size. The DD is small, so either SIMD or MIMD parallelism can be applied to the program; however, the maximum parallelism is 10,000 processors, assuming each PE executes the algorithm for one or more templates. Application of P processors will yield the speedup shown in VI. The output of this system is one word. It is imperative that the system keep up with the input; however, it is desirable to do such with a minimal cost.

The number of each calculation can be multiplied by the single-operand execution times of the tuples for each processor in the database. The sum of the products yields an approximate worst-case execution time for a single copy of each processor in the database to perform this algorithm. Actual execution time could be better due to clever software or special hardware func-

tions. For example, software that is written to ignore redundant calculations (e.g., calculating the address of b[j][k] only once in the expression: b[j][k]=b[j][k]+5). Also, by applying pipeline analysis techniques to this algorithm and using structural information about each processor, such as functional overlap, stages in processing pipelines, and the multiplicity of units, a more precise approximation of the single processor execution times can be obtained.

Based on the desired throughpt and response time, additional processors of the same type are repetitively added until a level composed of such processors could meet the time requirements. The number of processors is then multiplied by the cost of the associated hardware. To this amount, the price of other devices, such as memory and inter-processor communications links, is added to approximate the cost of the processing hardware involved. The processor chosen used for the design will be chosen based on the least expensive hardware.

Consider the application of a Motorola 68000 [20] to the above task. The tuples enumerating the operations and their respective times contains over 1000 instructions: a partial list is included for brevity:

{add r,#;add r1,r2;add (a)+,r;cond. branch; mov r,#;mov r,(a);mov #,(a);mul r1,r2; mul (a)+,r; sub r,#;sub r1,r2;sub (a)+,r}

where r stands for register, # stands for immediate, (a) stands for memory location stored in register "a", (a)+ stands for memory location stored in register "a" followed by incrementing "a."

The tuple describing the timings (in cycles) is:

{8,4,8,10(true)/8(false),8,12,12,70,74,8,4,8}

The 68000 has no functional overlap or pipelining other than a five stage instruction decoder. These tuples will be omitted. A 68000 has no special address calculation hardware, so a two-dimensional address calculation requires loading a register, multiplying by a memory location, and the addition of two memory locations. Assuming that the index variables are stored in registers and that fixed point numbers are stored in memory, a 12.5 MHz 68000 would take 1579 seconds to perform dynamic time warping on a single word. Using a multistage cube network that takes 1.0 msec for two transfers, 1600 processors in MIMD mode would take .998 seconds to perform dynamic time warping. (A thorough analysis should consider the overlap of CU and PE operations in SIMD mode; e.g., address calculations). Dynamic time warping is normally done with fewer than 100 reference templates because of its great computational complexity.

Such an analysis would be required for each processor in the database. Then, an actual implementation of the above approach would consider simulating the algorithm on the various processors to obtain a more accurate timing estimation. Finally, if no processor in the database could be used to implement this algorithm, the layer would need to be broken down into sub layers, each of which would be analyzed with the proposed techniques.

6. Conclusions

Using the above nine categories, an algorithm can be analyzed according to what requirements it places on a system. If many hardware components are analyzed and categorized according to abilities and processing times, a library containing information about these processors can be built. By using these models of algorithms and hardware to map the organization of each level in a multi-level design to a layer of software, computers can be used to aid in the design of systems for the specific needs of algorithms, thus making possible computer assisted design of special purpose parallel architectures.

In summary, this was a study of one approach to model the design of a class of macro-pipelined parallel architectures. Categories of hardware analysis were presented. Their relationship to the hardware requirements and their dependence on the algorithm to be performed was discussed. An example of the application of the parameters and tuples was shown. By studying approaches to bridging the gap between hardware and algorithms, computer aided special purpose machine design comes closer to being a reality.

Acknowledgements: The authors of this paper wish to extend their deepest thanks to M. Yoder, without whom the discussion on the isolated word recognition system could not have been done in real-time, and to M. Franklin, G.J. Lipovski, P. Swain, and A. van Tilborg, whose careful readings and comments helped to organize and clarify the ideas in this paper.

References

[1] Advanced Micro Devices Inc., *Am9511A Product Specification Sheet*, AMD Inc., Sunnyvale, CA, Mar. 1982.

[2] R.G. Arnold, R.O. Berg, and J.W. Thomas, "A modular approach to supersystems," *IEEE Trans. Comput.*, Vol. C-31, pp. 385-389, May 1982.

[3] A.J. Bernstein, "Analysis of programs for parallel processing," *IEEE Trans. Comput.*, Vol. EC-15, pp. 757-763, Oct..1966.

[4] E.C. Bronson and L.J. Siegel, "A parallel architecture for acoustic processing in speech understanding," *1982 Int'l. Conf. Parallel Processing*, pp. 307-312. Aug. 1982.

[5] G. DeMuth, "A distributed signal processor incorporating VLSI and high order language programming," *1983 Int'l. Conf. Acoustics, Speech, and Signal Processing*, pp. 439-442, Apr. 1983.

[6] M.J. Flynn, "Very high speed computing systems," *Proc. IEEE*, Vol. 54, pp. 1901-1909, Dec. 1966.

[7] W.K. Giloi, "Towards a taxonomy of computer architecture based on the machine data type view," *10th Annual Int'l. Symp. Comput. Arch.*, pp. 6-15, May 1983.

[8] M.J. Gonzalez, "Quantitative evaluations of distributed computing systems," *Rocky Mountain Symp. on Microcomputers: Systems, Software, and Arch.*, pp. 125-130, Aug. 1978.

[9] W. Handler, "The impact of classification schemes on computer architecture," *1977 Int'l. Conf. Parallel Processing*, pp. 7-15, Aug. 1977.

[10] W. Handler, "Standards, classification, and taxonomy; experiences with ECS," *IFIP Workshop on Taxonomy in Comput. Arch.*, pp. 39-75, June 1981.

[11] J.F. Hart, et al, *Computer Approximations*, John Wiley and Sons, Inc., New York, NY, 1968.

[12] R.W. Hockney and C.R. Jesshope, *Parallel Computer Architecture Programming, and Algorithms*, Adam Hilger Ltd, Bristol, 1981.

[13] R.W. Hueft and W.D. Little, "Improved time and parallel processor bounds for FORTRAN like loops," *IEEE Trans. Comput.*, Vol. C-31, pp. 78-81, Jan. 1982.

[14] K. Hwang and F.A. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill, New York, NY, 1984.

[15] S.I. Kartashev and S.P. Kartashev, "Dynamic architectures: problems and solutions," *Computer*, Vol. 11, pp. 26-42, July 1978.

[16] D.J. Kuck, Y. Muraoka, and S.C. Chen, "On the number of operations simultaneously executable in FORTRAN-like programs and their resulting speed up," *IEEE Trans. Comput.*, Vol. C-21, pp. 1292-1309, Dec. 1972.

[17] L.F. Lamel, L.R. Rabiner, A.E. Rosenburg, and J.G. Wilpon, "An improved endpoint detector for isolated word recognition," *IEEE Trans. Acoustics, Speech, and Signal Processing*, Vol. ASSP-11, pp. 777-785, Aug. 1981.

[18] G.J. Lipovski and A. Tripathi, "A reconfigurable varistructure array processor," *1977 Int'l. Conf. Parallel Processing*, pp. 165-174, Aug. 1977.

[19] J.D. Markel and A.H. Gray, Jr., "Fixed point truncation arithmetic implementation of a linear prediction autocorrelation vocoder," *IEEE Trans. Acoustics, Speech, and Signal Processing*, Vol. ASSP-22, pp.273-282, Aug. 1974.

[20] Motorola, *MC68000 16-bit Microprocessor User's Manual*, second edition, M68000UM(AD2), Jan. 1980

[21] K. Ogata, *Modern Control Engineering*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1970.

[22] C.V. Ramamoorthy and M.J. Gonzalez, "A survey of techniques for recognizing parallel processable streams in computer programs," *1969 Joint Comput. Conf.*, pp. 1-15, Aug. 1969.

[23] L.R. Rabiner, S.E. Levinson, A.E. Rosenburg, and J.G. Wilpon, "Speaker-independent recognition of isolated words using clustering techniques," *IEEE Trans. Acoustics, Speech, and Signal Processing*, Vol. ASSP-27, pp. 336-349, Aug. 1979.

[24] L.R. Rabiner and R.W. Schafer, *Digital Processing of Speech Signals*, Prentice-Hall, Englewood Cliffs, NJ, 1978.

[25] M.R. Sambur and L.R. Rabiner, "A speaker-independent digit-recognition system," *Bell System Technical Journal*, Vol. 54, pp. 81-102, Aug. 1975.

[26] H.J. Siegel, *Interconnection Networks for Large-Scale Parallel Processing: Theory and Case Studies*, Lexington Books, Lexington MA, 1985.

[27] H.J. Siegel, P.H. Swain, and B.W. Smith, "Parallel processing implementations of a contextual classifier for remote sensing data," *1980 Machine Processing of Remotely Sensed Data Symp.*, pp 10-28, June 1980.

[28] L.J. Siegel, H.J. Siegel, and P.H. Swain, "Performance measures for evaluating algorithms for SIMD machines," *IEEE Trans. Software Eng.*, Vol SE-8, pp. 319-331, July 1982.

[29] L.J. Siegel, H.J. Siegel, P.H. Swain, G.B. Adams III, W.E. Kuhn III, R.J. McMillen, T.A. Rice, K.D. Smith, and D.L. Tuomenoksa, "Distributed computing for signal processing: modeling of asynchronous parallel computation 1983 progress report," *Technical Report TR-EE 83-11*, E.E. School, Purdue Univ., West Lafayette, IN 47907, Mar. 1983.

[30] B.W. Smith, H.J. Siegel, and P.H. Swain, "Contextual classification on a CDC flexible processor system," *1981 Machine Processing of Remotely Sensed Data Symp.*, pp. 283-291, June 1981.

[31] H.S. Stone, "Problems of parallel computation," in *Complexity of Sequential and Parallel Numerical Algorithms*, ed. J.F. Traub, Academic Press, New York, NY, 1973.

[32] H.S. Stone, "Parallel computers," in *Introduction to Computer Architecture*, 2nd edition, ed. H.S. Stone, Science Research Associates, Chicago, IL, 1980, pp. 363-425.

[33] P.H. Swain, H.J. Siegel, and B.W. Smith, "Contextual classification of multispectral remote sensing data using a multiprocessor system," *IEEE Trans. Geoscience and Remote Sensing*, Vol. GE-18, pp. 197-203, Apr. 1980.

[34] D.L. Tuomenoksa, G.B. Adams, H.J. Siegel, and O.R. Mitchell, "A parallel algorithm for contour extraction: advantages and architectural implications," *IEEE Comput. Society Conf. Comput. Vision and Pattern Recognition*, pp. 336-344, June 1983.

[35] C.R. Vick, *A Dynamically Reconfigurable Distributed Computing System*, Ph.D. Dissertation, E.E. Dept., Univ. of Alabama, Auburn, AL, 1979.

[36] C.M. Woodside and D.W. Craig, "Function allocation in a tightly coupled signal processing system," *4th Int'l. Conf. Distributed Computing Systems*, pp. 118-125, May 1984.

[37] M.A. Yoder and L.J. Siegel, "Dynamic time warping algorithms for SIMD machines and VLSI processor arrays," *1982 IEEE Int'l. Conf. Acoustics, Speech, and Signal Processing*, pp. 1274-1277, May 1982.

END

FILMED

6-86

DTIC